A Serverless, Wide-Area Version Control System

by

Benjie Chen

B.S., Massachusetts Institute of Technology (2000)
M.Eng, Massachusetts Institute of Technology (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2004

Author .....
Department of Electrical Engineering and Computer Science
May, 2004

Certified by .....
Robert T. Morris
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Serverless, Wide-Area Version Control System

by

Benjie Chen

Submitted to the Department of Electrical Engineering and Computer Science
on May, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

This thesis describes Pastwatch, a distributed version control system. Pastwatch maintains versions of users' shared files. Each version is immutable: to make changes, a user checks out a version onto the user's computer, edits the files locally, then commits the changes to create a new version. The motivation behind Pastwatch is to support wide-area read/write file sharing. An example of this type of sharing is when loosely affiliated programmers from different parts of the world collaborate to work on open-source software projects.

To support such users, Pastwatch offers three properties. First, it allows users who travel frequently or whose network connections fail from time to time to access historical versions of the shared files or make new versions while disconnected. Second, Pastwatch makes the current and historical versions of the shared files highly available. For example, even when their office building experiences a power failure, users can still create new versions and retrieve other users' changes from other locations. Supporting disconnected operation is not adequate by itself in these cases; users also want to see others' changes. Third, Pastwatch avoids using dedicated servers. Running a dedicated server requires high administrative costs, expertise, and expensive equipment.

Pastwatch achieves its goals using two interacting approaches. First, it maintains a local branch tree of versions on each user's computer. A user can check out versions from the local tree and commit changes into the local tree. Second, Pastwatch uses a shared branch tree in a DHT to publish users' new versions. It contacts the tree to keep a user's local branch tree up-to-date.

Thesis Supervisor: Robert T. Morris
Title: Associate Professor

"Learn to be comfortable at the uncomfortable."

Daniel Cogan-Drew

Acknowledgments

My studies have been sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933, by grants from NTT Corporation under the NTT-MIT collaboration, by a scholarship from the USENIX Association, and by a NSF Career Grant. I am grateful for these supports.

The Ivy peer-to-peer file system, joint work with Robert Morris, Athicha Muthitacharoen, and Thomer Gil, inspired much of the thesis. The design and implementation of Aqua is joint work with Alex Yip, borrowing source code from the DHash/Chord project. Most of my programs use the SFS tool-kit, by David Mazières, for event-driven programming and RPC.

Butler Lampson, Barbara Liskov, and Robert Morris spent valuable time and effort helping me improve this thesis. Their patience astonish me. I would also like to thank members of PDOS, and Alex Yip in particular, for reading drafts, offering comments, and discussing ideas.

I have had the privilege to work with many amazing individuals at MIT; too many to name. Specifically, I would like to thank past and present members of PDOS for their subtle expectations, Robert Morris and Frans Kaashoek for advising me in the past few years, and Chuck Blake, Rodrigo Rodrigues, and Max Poletto for bring balance to my world at MIT.

Playing on the MIT Ultimate team has changed my life for the better. Dan Cogan-Drew, my coach, has taught me to always have high expectations and demand more of myself. My teammates, Tom Liptay, Shawn Kuo, and Peter Mayer in particular, have taught me how to have fun and be intense at the same time. I will treasure my time with them for the rest of my life.

Robert Morris, my advisor, has shown an incredible amount of faith in me. I have learned more from him than I can ever repay. I aim to hold myself to his high standards wherever I go.

My family has always supported me. Among many things, my parents have taught me how to learn and how to be a good person. I hope one day I can love and care for my children as much as they love and care for me. This thesis is dedicated to them.

Most importantly, I am incredible lucky to be spending my life with Melina. Her love, care, patience, and enthusiasm keep me going. She makes me a better person; I hope I can do the same for her.

Contents

List of Figures

List of Tables

Chapter 1

Pastwatch: Goals and Design Overview

This thesis describes Pastwatch, a distributed version control system. Pastwatch maintains versions of users' shared files. Each version is immutable: to make changes, a user checks out a version onto the user's computer, edits the files locally, then commits the changes to create a new version. The motivation behind Pastwatch is to support wide-area read/write file sharing. An example of this type of sharing is when loosely affiliated programmers from different parts of the world collaborate to work on open-source software projects.

To support such users, Pastwatch offers three properties. First, it allows users who travel frequently or whose network connections fail from time to time to access historical versions of the shared files or make new versions while disconnected. Second, Pastwatch makes the current and historical versions of the shared files highly available. For example, even when their office building experiences a power failure, users can still create new versions and retrieve other users' changes from other locations. Supporting disconnected operation is not adequate by itself in these cases; users also want to see others' changes. Third, Pastwatch avoids using dedicated servers. Running a dedicated server requires high administrative costs, expertise, and expensive equipment.

1.1 Current Practices Are Inadequate

Many collaborative projects use CVS [4] for version control. CVS puts version histories of the shared files into a repository on a server and exposes the repository to networked users through a networked file system or a remote execution protocol (e.g. ssh [47]). Typically, a project member

copies the shared files from a CVS repository to the user's local disk, makes changes to these local files, and then contacts the repository later to commit changes. CVS automatically merges non-conflicting changes and reports conflicts before committing changes. CVS maintains a version history for each file so that earlier versions of the shared files are readily available.

Using a dedicated server for version control poses some problems over the wide-area network. First, a user may want to access the repository even if the user cannot reach the server. This scenario could arise, for example, if either the user or the server's network connection does not work or when the server crashes. Second, using a single server means that all users must trust the server, and, more prohibitively, that the server's administrator must be willing to grant access to the server to users from different organizations. These types of trust may not exist between a project's loosely affiliated users. Finally, users may not be technically skilled enough or want to spend the resources to maintain a reliable server. The popularity of SourceForge,[1] a third-party service, supports this claim: over 80,000 open-source software projects use SourceForge to host their shared files.

Despite its popularity, SourceForge is not an ideal solution. First, users cannot access a repository if they are disconnected. Second, despite SourceForge's continuing efforts to upgrade its network and computing facilities to keep up with the increasing number of users, its service frequently suffers from server overload, network congestion, and software and hardware failures. For example, in January of 2004, SourceForge repositories were unavailable for at least 40 hours. At other times, users have experienced unacceptably slow service.

Some version control tools, such as Arch[2] and BitKeeper[3], improve availability by replicating a shared repository on each user's machine. A user can access and modify the repository using only the local replica, but users still depend on a dedicated server to store and publish their changes.

1.2 What Pastwatch Users See

Pastwatch performs version control at the granularity of projects. It maintains versions of each project's shared files. Pastwatch organizes these versions using a *branch tree*. Each node on the tree is a version. A version is immutable, contains a copy of all of the files, and *refers to its parent on the tree*. The branch tree captures commit operations: each node is a version, and a child on the tree results when a user commits changes to the files in the parent. A fork in a tree (i.e. a parent with

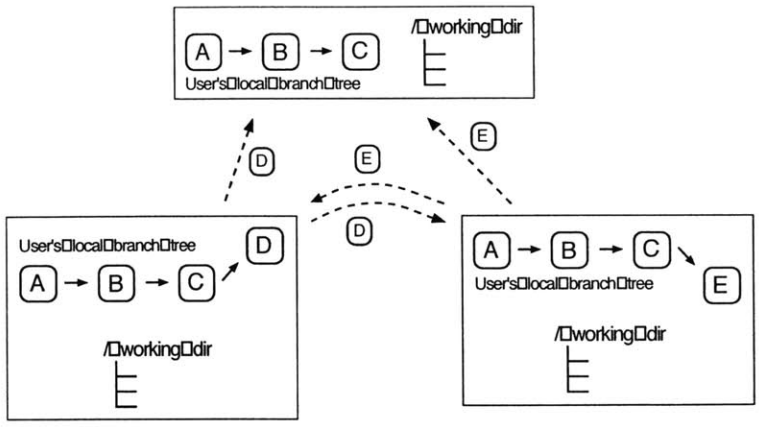

Figure 1-1: Pastwatch's user model. Each rectangular box is a user's computer. Each user has a local branch tree of versions (oval boxes) and can check out a version from the tree into a working directory. When a user commits changes, Pastwatch extends the user's branch tree. A synchronization mechanism propagates new versions between users. This mechanism is asynchronous; a new version added to one branch tree may not appear in another branch tree immediately.

two children) reveals when users commit changes to the same parent.

To support disconnected operation, Pastwatch maintains multiple branch trees for each project, *one on each user's computer.* See Figure 1-1. To make changes to the shared files, a user checks out a version from the local branch tree into a working directory, makes changes locally in the working directory, then commits the changes. The commit operation creates a new version on the user's local branch tree; the new version is a child of the version that was checked out into the working directory. To avoid leaving a fork on the local branch tree at the time of the commit operation, Pastwatch refuses to commit changes if the working directory was not checked out from a leaf of the local branch tree.

Pastwatch propagates new versions among users' branch trees. The propagation mechanism is asynchronous; Pastwatch tries to make a user's new version visible to other users immediately, but it may not always be able to do so. This means a user may commit changes when the local branch tree is stale. In this case, the user's new leaf version may share a parent with another user's new version. These versions form a fork in any tree to which they have been propagated. *The fork correctly reveals the fact that Pastwatch allowed a user to commit changes without knowing another user's newest version.* Pastwatch offers tools to help users detect and recover from forking: a user can merge changes from one branch of the tree into another and mark the first branch as inactive. In

practice, users with network connectivity rarely leave forks on their branch trees.

While Pastwatch cannot guarantee that a user's committed changes become visible to other users immediately, it does provide *eventual consistency*: if all disconnected users re-connect and users do not commit new changes, then eventually all users' branch trees become identical.

1.3 Design Overview

To propagate versions among users' local branch trees, Pastwatch stores the versions in a shared branch tree in a distributed hash table (DHT) [42]. To bring a user's local branch tree up-to-date, Pastwatch scans this tree and adds new versions to the local tree. Using a DHT has multiple benefits. First, if users are connected, DHT hosts do not fail, and network delays are low, a DHT can provide the appearance of a "server". Once a user adds a version to the DHT branch tree, other users can contact the DHT and download the new version. Second, a DHT replicates data across multiple hosts. This not only improves the likelihood that a user's new changes can be downloaded by other users, but also avoids dedicated servers.

In the simplest case, a project may choose to set up a DHT with only the project members' computers. Alternatively, many projects could share a public DHT [20]. For example, for open-source software projects, a public DHT could consist of voluntary hosts that already donate network and storage resources to distribute open-source software.

Pastwatch synchronizes a user's local branch tree against the DHT branch tree using three operations, as shown in Figure 1-2. In this figure, a dashed box is the result of an operation represented by the nearby dotted arrow. First, when a user commits changes, Pastwatch adds a new version, E, to the user's local branch tree using `extendLocal`. Before the commit operation ends, it calls `extendDHT` to add E to the DHT branch tree as well. Pastwatch periodically adds new versions from the DHT branch tree to a user's local branch tree using `updateLocal`. For example, for the user in the top of the figure, Pastwatch adds version E to that user's branch tree. To increase the likelihood that a user can immediately see other users' new versions, Pastwatch also calls `updateLocal` before every user operation (e.g. check out).

Each version on the DHT branch tree is immutable; Pastwatch guarantees that once created, a version's parent and contents cannot change. This means that if a version appears on two users'

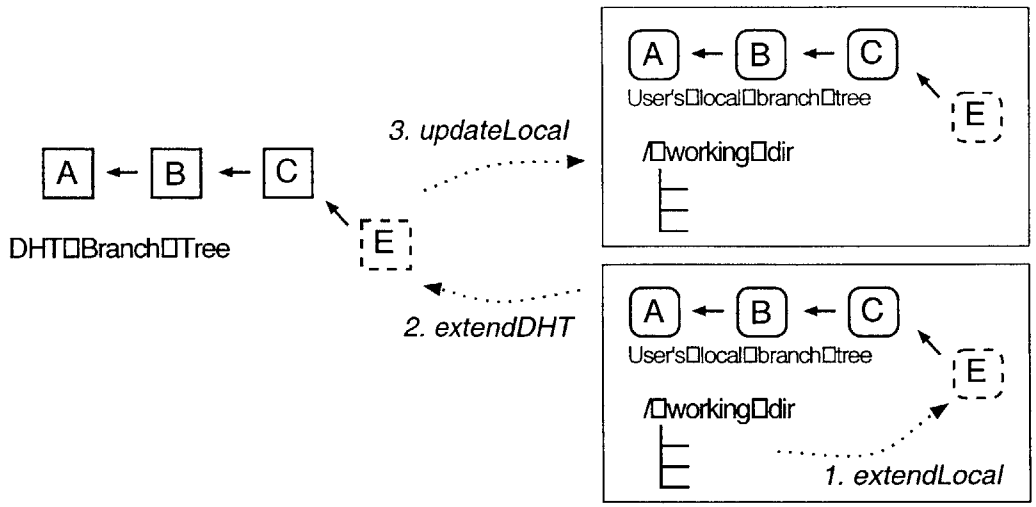

Figure 1-2: Interactions between a DHT branch tree (left) and users' local branch trees (right). A dashed box is the result of an operation represented by the nearby dotted arrow. A commit operation first adds a new version, E, to the user's local branch tree using `extendLocal`. It then adds E to the DHT branch tree using `extendDHT`. Finally, Pastwatch copies E fro the DHT branch tree into another user's local tree using `updateLocal`.

local branch trees, they have the same parent on both trees.

Pastwatch does not depend on the DHT to behave in a consistent manner. The shared DHT branch tree helps users propagate their changes to each other. When the DHT provides atomicity, users can find each other's changes promptly. Otherwise, there may be a delay. If an inconsistency persists, a user's Pastwatch software re-inserts versions that the user had inserted earlier, but are now missing from the DHT.

Challenges and Solutions

The DHT branch tree allows different users' Pastwatch software to find other users' new versions. One way to build this tree is to store each version in the DHT as an immutable block. The tree structure is reflected by the fact that each version also refers to its parent on the branch tree. A mutable block would point to the leaves of the tree. When users add new leaves to the DHT tree, they update this mutable block.

Unfortunately, most DHTs do not provide atomicity; a user may not be able to fetch what another user has just inserted into the DHT. This means using multi-writer blocks could result in

inconsistencies. For example, suppose after a user updates the mutable block to include a new leaf, another user fetches a stale copy of the block that does not point to the new leaf. If the second user modifies the stale copy to include yet another new leaf and inserts the resulting copy of the block into the DHT, the first user's new leaf would be lost.

Pastwatch copes with the lack of atomicity by only using single-writer DHT blocks. It implements the DHT branch tree using a set of logs, one per user. Each user's log contains all the versions that the user has created and uses a mutable *log-head* to point to the log's contents. Only one user modifies each mutable log-head. Pastwatch scans all the logs to construct the DHT branch tree: each version in a log points to both the parent version on the DHT branch tree and the previous version in the log. Per-user logging was first developed for the Ivy peer-to-peer file system [29]. Ivy and Pastwatch use the technique to build different systems, however.

1.4 Implementation Goals

Several implementation goals make the system more practical in the wide-area. In this environment, loosely affiliated users and contributors to the DHT may not trust each other.

- Users should be able to recover from any unwanted changes committed by a malicious user or an intruder pretending to be a user after compromising that user's computer.

  Pastwatch meets this goal because a malicious user can only add new versions to another user's branch tree. This means bad changes in a new version could be rolled back by using states from the version's parent. Pastwatch also guarantees that a version's name is unique and cannot be re-bound to a different version.

- Users may not always trust the DHT. Pastwatch should cope with an untrusted DHT and help users detect the DHT's attempts to forge or hide changes.

  Pastwatch uses cryptographic techniques to verify the integrity of the data it retrieves from the DHT. It uses a branch tree to detect inconsistencies due to the DHT hiding changes from the user. If the DHT hides a new version from some users (i.e. the victims) but not the others, the victims and other users' new versions will form two branches. If the attack terminates, a fork on the users' branch trees reveals the inconsistent behavior. If the attack persists, the

victims would not be able to see any of the versions from the other branch. In this case, off-line communication (e.g. an e-mail that says "did you see my changes to file X?") reveals the attack. This is similar to fork-consistency [28].

A network partitioning among DHT hosts also gives the appearance of a stale-data attack. By itself, Pastwatch cannot distinguish a short-term attack from a network partitioning; in any case, if a DHT often behaves in an inconsistent manner, users may want to switch to a new DHT with a new set of hosts.

- The feasibility of Pastwatch's design depends on the premise that users would want to cooperatively store data for each other. For legal, political, or personal reasons, some contributors to the DHT may want to restrict what can be stored on their computers. To make its cooperative vision more practical and feasible, Pastwatch helps these contributors audit data stored on their servers and remove blocks they do not want to store. If a host refuses to store a project's blocks, the project needs to find a different DHT. Without Pastwatch's help, these tasks are difficult to implement in a DHT. Most DHTs do not support these features.

1.5 Contributions

This thesis has several contributions.

- It describes how to provide eventual consistency for an optimistically replicated object (i.e. a project's shared files) using branch trees. Each user organizes every users' changes to the object into a local branch tree. Each node on the tree is an immutable version of the object, refers to its parent, and contains exactly the result of applying a set of changes to the parent's contents. Given the same set of user changes, every user independently constructs the same branch tree.

- It uses forks on a branch tree to reveal when a user made changes to a stale local replica and created a new version. This could occur when some users are disconnected or when the propagation mechanism does not make a user's new version visible to other users immediately. Pastwatch also helps users detect and recover from forking.

- It describes how to build a single-writer, multi-reader data structure (i.e. the branch tree) in the DHT using per-user logging. Each user appends changes to the data structure (i.e. new versions) to the user's own log and scans all the logs to reconstruct the data structure. This arrangement is attractive because most DHTs require mutable blocks to be cryptographically signed and users may not want to share private-keys.

- It uses forks on a branch tree to reveal when a DHT behaves in an inconsistent manner, such as returning stale data or no data to users. These inconsistencies could be caused by long network delays, host failures, networking partitioning, or malicious DHT hosts.

1.6 Thesis Overview

The rest of the thesis is organized as follows. Chapter 2 describes Pastwatch's user model and semantics of user-visible operations. Chapter 3 describes how Pastwatch implements the user model. Chapter 4 describes Pastwatch's implementation. Chapter 5 outlines the design and implementation of Aqua, including how Pastwatch helps Aqua hosts perform garbage collection, access control, and audit. Chapter 6 evaluates the system. Chapter 7 describes related work. And Chapter 8 concludes.

Notes

[1] www.sourceforge.net
[2] wiki.gnuarch.org
[3] www.bitkeeper.com

Chapter 2

Pastwatch User Model

Pastwatch offers version control at the granularity of projects. A project has a set of *users* who can make changes to the project's shared files. One user is a *project administrator*, who creates the project. The state of each project is a set of *branch trees*, one for each user. The nodes on each tree are *versions*. Each version contains all of the shared files and directories. A child version contains changes to the files in the parent version.

Initially, each user's branch tree for a new project has an empty version – the root of the tree. Using operations that Pastwatch provides, users can modify the project's shared files by creating subsequent versions. A user only adds new versions to the user's own local branch tree. Pastwatch uses a synchronization mechanism to propagate new versions among all the branch trees.

The project administrator can add and remove users from the project's list of users. Each user also has a name that is unique within the project. There is access control to prevent non-project members from modifying the project's tree. This chapter ignores these issues.

This chapter describes Pastwatch's user model in terms of operations that can be done on the branch trees, the synchronization mechanism, and the user-visible semantics.

2.1 Branch Tree Operations

A branch tree consists of a set of versions. There is an initial, empty version in each tree. Each version contains the following:

- **User.** Each version contains the name of the user who created the version.

- **Name.** Each version has a user-proposed, human-readable name.
- **Contents.** Each version contains the states of all the shared files and directories.
- **Parent.** With the exception of the initial, empty version that is the root of the tree, each version has a parent. A child version contains some changes that a user has made to the states of the shared files and directories in the parent version.
- **List of superseded versions.** A version can *supersede* one or more other versions. This list could be empty. Version `v` superseding version `s` is a hint to the users that `v` contains merged states of `s` and `v`'s parent.

Each version is immutable; there is no way to modify any of a version's data once it is created. *Each branch tree is append-only*; there is no way to modify an existing portion of the tree. Each version also has an unique *version identifier*. This identifier can be computed deterministically from the version's data.

Two users could propose the same name for two different versions. In this case, Pastwatch switches to using their unique version identifiers to refer to the versions. In this sense, the true name of a version is a combination of the version identifier and the user-proposed name.

The following functions on a branch tree are useful.

- `getLeaves(v)`: returns a set of versions S such that each version in S is a leaf version, is a descendant of version `v`, and is not superseded by any other version.
- `getPrev(v)`: returns version `v`'s parent on the tree. If `v` is the root, signals error.
- `getAllLeaves(v)`: returns a set of versions S such that each version in S is a leaf version and is a descendant of version `v`.

This thesis refers to a sequence of versions between the root of a branch tree and a leaf version as a *branch*. Through Pastwatch, a user can obtain a list of all the branches that have not been superseded using `getLeaves`. *If a version supersedes another version* `s`*, then all the ancestors of* `s` *that are on a single branch in the tree are also superseded.* A branch is superseded if its leaf is superseded. For example, in Figure 2-1, the dotted line from G to F shows that G supersedes F; G contains merged states of both D and F. In this case, `getLeaves` only returns G, and the user

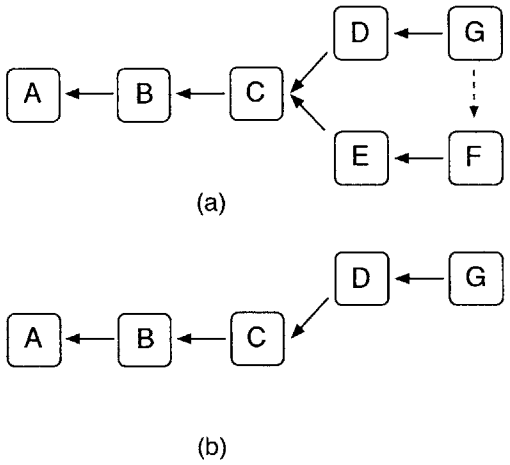

Figure 2-1: An example of a branch tree. Each version refers to its parent (the solid arrow) and possibly a list of other versions it supersedes (the dotted arrow in part a). The solid arrows form the tree. Part b shows versions that a user can reach using `getLeaves` and `getPrev`. `getLeaves` returns leaf versions that have not been superseded.

is only aware of one branch between A and G. Note that a user could specifically ask Pastwatch to return the results of `getAllLeaves`, which would reveal two branches.

The following procedures manipulate a project and a user's branch tree for that project.

- `createProject(admin,users)`

  Creates a new project with `admin` as the public-key of the administrator and `users` as the public-keys of users who can access and modify the project's tree. The project's ID is the result of evaluating some deterministic function `H(admin)` and should be distinct from results computed for other administrator public-keys. This operation fails if `H(admin)` already exists as a project ID.

- `fetchTree(u,projID)`

  Returns user `u`'s branch tree if `projID` is a valid project. Otherwise, signals error.

- `extend(u,projID,user,name,parent,supersede,contents)`→`version`

  Uses user `u`'s branch tree for project `projID`. If `parent` or a version in `supersede` is not in the tree, signals error.

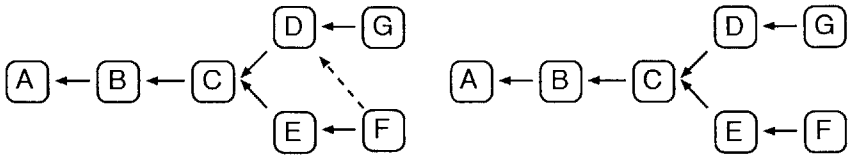

Figure 2-2: An example of a branch tree (left) and what a user sees (right). The dotted arrow shows that F supersedes D. But because G extends D, both G and D are still visible when a user calls `getLeaves` and `getPrev`.

> Otherwise, creates version `v` with `user` (name of the user who first created the version), `name`, `parent`, `supersede`, and `contents`, adds `v` to the project's tree, and returns `v`. This operation is deterministic in that given the same `user`, `name`, `parent`, `supersede`, and `contents`, it returns the same version.

Only `extend` can create a version. The `extend` procedure preserves an invariant that if a version `p` supersedes `s`, than `s` cannot be `p` or `p`'s parent.

`extend` allows a new version to use a superseded version as its parent. In this case, `getLeaves` may report the new version if it is a leaf. Figure 2-2 shows an example of a branch tree with a superseded version as a parent (on the left) and what users see when they use `getLeaves` and `getPrev` (on the right).

A user does not directly modify the local branch tree. Instead, the user can check out a leaf version from the tree into a *working directory* on the user's computer, make changes locally, then commit the changes back into the branch tree. The commit operation calls `extend` to add a new version on the tree. The new version contains states of the files in the working directory and uses the version that the working directory was checked out from as its parent.

2.2 Synchronizing Branch Trees

Pastwatch uses a synchronization mechanism to propagate new versions among users' branch trees. See Figure 2-3. The mechanism is asynchronous. After Pastwatch uses `extend` to add a new version to a branch tree, another user may not be able to see the version immediately, or ever. This asynchronous propagation model suggests that Pastwatch does not distinguish connected users from disconnected users; both sets of users may not be able to see other users' new versions immediately.

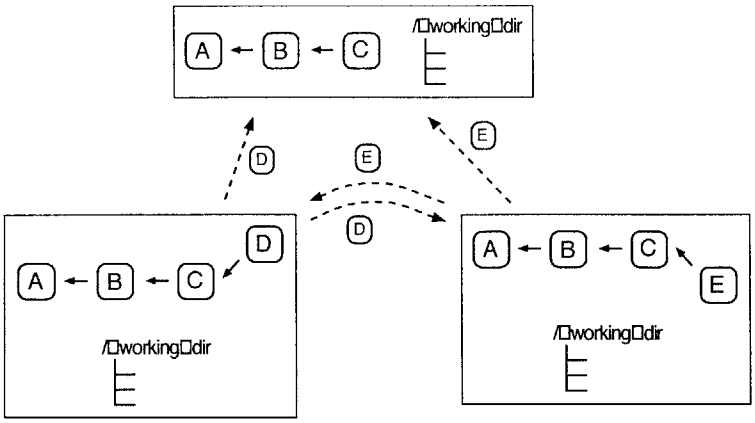

Figure 2-3: Pastwatch's user model. Each rectangular box is a user's computer. Each user has a local branch tree (oval boxes) and can check out a version from the tree into a working directory. When a user commits changes from a working directory, Pastwatch extends the user's branch tree. A synchronization mechanism propagates new versions between users. This mechanism is asynchronous. A new version added to one branch tree may not appear in another branch tree immediately, or ever.

In practice, Pastwatch can usually propagate new versions among connected users promptly.

The synchronization mechanism does not add a version `v` to a user's local branch tree unless `v`'s parent and all the versions that `v` supersedes are already in the branch tree. That is, suppose a *consistent branch tree* is one in which each version's parent and superseded versions also exist in the tree. The synchronization mechanism preserves the consistency of each user's local branch tree.

The synchronization mechanism uses `extend` to add versions from one user's branch tree to the branch tree of another user. Assume the user is `u` and the project ID is `P`. For each version `v`, Pastwatch calls `extend(u, P, v.user, v.name, v.parent, v.supersede, v.contents)`. Each `extend` succeeds because its parent and superseded versions must already be on the local tree, and its supersedes list must be well-formed since `v` was added to another branch tree, by another user's Pastwatch software, using `extend` as well.

The fact that each immutable version refers to its parent means users' branch trees are consistent. That is, if a version `v` appears on two users' branch trees, then `v` has the same parent on both trees. Furthermore, all the versions between `v` and the root of one tree appear on the other tree.

The synchronization mechanism tries to propagate all versions from one users' branch tree to each of the other users' branch trees. The state for a project at any time is a logical tree that is the union of all the users' branch trees. If this tree is t_a at one point and users do not subsequently add

new versions to their trees, then eventually, if a user receives all the new versions from the other users, that user's branch tree will equal to t_a.

2.3 Semantics of User-visible Operations

A user issues the following operations to access/modify the local branch tree. Assume that Pastwatch can obtain a copy of the local branch tree using `fetchTree(projID)`.

- **checkout**: check out a version from the user's branch tree and creates a working directory.
  - Optional argument `v`. If `v` is not a version on the tree, fail. If no argument is given, call `S=getLeaves(r)`, where `r` is the root of the tree. If `S` returns more than one version, output `S` and terminate. Otherwise, use the version in `S` as `v`. Most users will issue checkout without an argument.
  - Create a new working directory. The resulting working directory is similar to a version on the branch tree. Specifically, it contains the contents of `v`, a *working version* that can be considered as the parent of the working directory, and a *supersede list*. After the checkout, the working version is `v` and the supersede list is empty.
- **latest**: return the result of `getLeaves(v)`, where `v` is a working directory's working version. Can optionally return the result of `getAllLeaves(v)`.
- **update**: update working directory to a leaf that is the descendant of the working version.
  - Assume the working version is `v0`. Call `S=getLeaves(v0)` on the local branch tree. If `S` contains more than one version, output `S` and terminate. If `S={v0}`, report which files contain un-committed changes and terminate. If `S` is empty, then `v0` has been superseded. In this case, report the result of `getLeaves(r)`, where `r` is the root of the tree, and terminate.
  - Assume `S={v1}`. Combine the contents of the working directory and `v1`. Put the combined states in the working directory. Change the working version to `v1`.
  - If `v1` contains changes that conflict with an un-committed change in the working directory, leave the conflicting changes in the affected files and report the names of these

files. Pastwatch expects the user to fix conflicts manually. A user can only commit changes if all the conflicts have been fixed.

- **merge**: merge contents of a version from a different branch into the working directory, without changing the working version.

  - Argument `v1` is not a descendant or ancestor of the working version. Fail otherwise.
  - Combine the contents of the working directory and `v1`. Put the combined states into the working directory. Report any conflicts that may arise and leave the conflicts in the affected files in the working directory.
  - Unlike update, leave working version unchanged. Record in the working directory's supersede list. A user may issue merge multiple times in a working directory.

- **commit**: commit changes from a working directory.

  - Fail if a file in the working directory contains conflicts, if the working version `v0` has been superseded, or if `v0` is not a leaf on the local branch tree. The last restriction prevents users from leaving forks on their trees. For example, it does not allow a user from committing from two working directories with the same working version, one commit after another.
  - Assume the user is `u`. Call `extend(u,projID,u,name,v0,s,c)` on the local branch tree, where `name` is a new version name that Pastwatch generates, `s` is the working directory's supersede list, and `c` is the contents of the working directory.

The main difference between Pastwatch and CVS [4] is that a user without network connectivity may commit changes. As a result, the new version may not appear in other users' branch tree until much later. This means that on a commit operation, when Pastwatch verifies that the working version is a leaf on the user's branch tree, it cannot guarantee that the working version is a leaf on every user's branch tree. Therefore, users could *implicitly* create branches of the shared files.

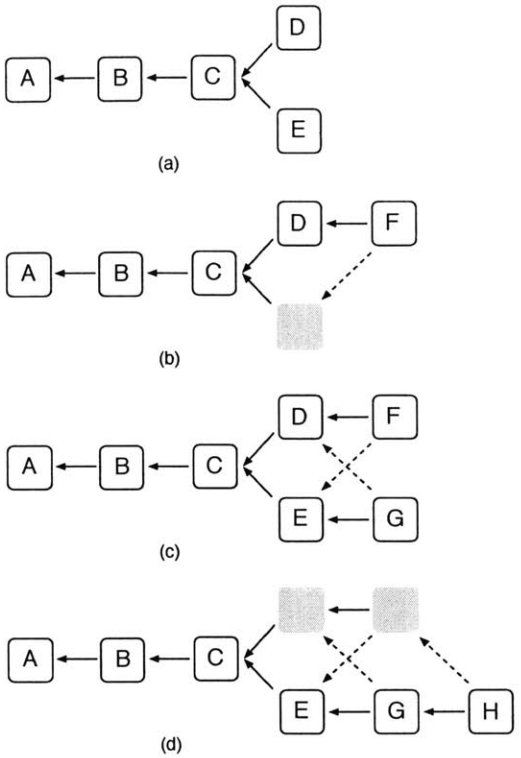

Figure 2-4: Examples of how users can merge changes between branches. See text in Section 2.3.1.

2.3.1 Fixing Forks

A user can "remove" a fork with the following steps. Figure 2-4-(a) shows what the user's branch tree looks like before running these operations.

1. Check out D into a working directory.
2. In that working directory, issue the merge operation with E as the argument.
3. Fix conflicts in the working directory, if any.
4. Commit changes from the working directory. The commit operation adds a new version F to the user's branch tree as a child of D. F also supersedes E, so the user no longer see E using `getLeaves`. See Figure 2-4-(b).

Two users may run these steps concurrently and may not agree on which version should be superseded and each chooses a different version. As a result, when their new versions propagate

to each other, each user may see a branch tree that looks like Figure 2-4-(c). A user can see both branches using `getLatest`, since each branch has a leaf that has not been superseded. A user can run the above steps again from a working directory with G as the working version. This results in Figure 2-4-(d).

Chapter 3

Pastwatch Design

To synchronize users' local branch trees, Pastwatch stores a shared branch tree in a distributed hash table (DHT). Pastwatch periodically tries to contact the DHT to bring a local branch tree up-to-date and to add new versions to the DHT branch tree. Specifically, for a user with network connectivity, this occur before every user operation and at the end of a commit operation.

Pastwatch uses a DHT because under the right conditions (which are the common case), a DHT has the appearance of a central server: once a user extends the DHT tree, another user can retrieve the new version from the tree immediately. A DHT also replicates data onto multiple hosts, which increases the likelihood that a user's changes can be found by others and avoids dedicated servers. Furthermore, typical DHT systems do not require participating hosts to trust each other. This property makes a DHT easy and practical to deploy over the WAN, across multiple organizations.

Pastwatch does not rely on the DHT for consistency or reliability. The shared branch tree in the DHT helps users propagate their changes to each other. When the DHT provides atomicity, users can find each other's changes promptly. Otherwise, there may be a delay. If the DHT lose some data, users can always re-insert them from their local branch trees.

Unless otherwise stated, assume that users and DHT hosts are not malicious.

3.1 Project and Users

Pastwatch offers version control at the granularity of projects. There is a *membership list* for each project. The membership list is created, cryptographically signed, and distributed by a project's

administrator. The membership list contains an entry for each user. Each entry has

- A user name that is unique within the project.
- Hash of the user's public-key. Pastwatch uses this key to authenticate users' data so that only project members can commit new versions (see Section 3.3.2).

The ID of each project is the hash of the public-key that can be used to verify the membership list's signature. This mechanism binds a project to an administrator. This binding is intentional. To change administrator, a new administrator would need to sign and distribute a new list for a new project. The new project may inherit the states of the previous project. Pastwatch makes users explicitly aware of which administrator controls the project.

A Pastwatch user is a human user's identity on one computer. If a human user has multiple computers, multiple Pastwatch user identities are needed, one for each computer. The membership list summarizes the users and their corresponding user names. Each user must have an unique name in the project.

3.2 Naming

Pastwatch assigns a human-readable name, of the form `user:seqn`, to each version. `user` is the name of the user who first created the version, and `seqn` is that user's per-user sequence number. Each user's sequence number starts at 1 and increments for each new version that user creates. These names are unique within each project.

3.3 Using a DHT

To synchronize users' local branch trees, Pastwatch stores a shared branch tree in a DHT. Users can retrieve other users' new versions from this tree and add new versions onto this tree. A user's computer does not need to be part of the DHT, but it should know how to contact a host in the DHT.

3.3.1 The DHT Abstraction

A DHT stores key-value pairs, called blocks. Each block has an identifier k, which users can use to retrieve the block's value v. The DHT offers the following simple interface: `put`$(id, value)$ and

`get(id)`. This thesis refers to the identifier of a DHT block as its GID.

A DHT offers two types of *self-authenticating* blocks. A *content-hash block* requires the block's GID to be the SHA-1 [13] hash of the block's value. A *public-key block* requires the block's GID to be the SHA-1 hash of a public-key and the value to be signed using the corresponding private-key. Each public-key block's value includes the public-key and a version number. The writer of a public-key block increments this version number on every write. The DHT overwrites an existing public-key block only if the new block's version number is higher than that of the existing block.

Assumptions about DHT Semantics

Pastwatch does not assume an atomic DHT. By observation, many existing DHTs [11, 12, 26, 38] do not offer atomicity. A `get` may not always return the value of the latest `put`. Worse yet, a user may not be able to retrieve a block that the user or another user was able to retrieve earlier. Atomic operations are challenging to implement in a DHT because of dynamic membership, host failures, long network delays, and potentially network partitioning among DHT hosts. Some systems provide atomicity [24, 35], but many choose not to do so to avoid complexity.

3.3.2 Branch Tree Structure

Structurally, the DHT branch tree looks like a user's local branch tree. Pastwatch uses *commit records* to represent versions on the DHT branch tree. It stores commit records as content-hash blocks. Each commit record contains

- **User.** Name of the user who created the commit record.
- **Name.** Each commit record has a human-readable name for the version.
- **Contents.** A commit record has the states of the shared files in the corresponding version.
- **Parent.** Each commit record contains the GID of its tree parent.
- **Supersede list.** A commit record lists the GIDs of other commit records on the DHT branch tree that it supersedes.

Directly storing a shared branch tree (e.g. the one in Figure 3-1-a) in the DHT is challenging. Users would need to modify some shared mutable DHT block that contains GIDs of the leaves of

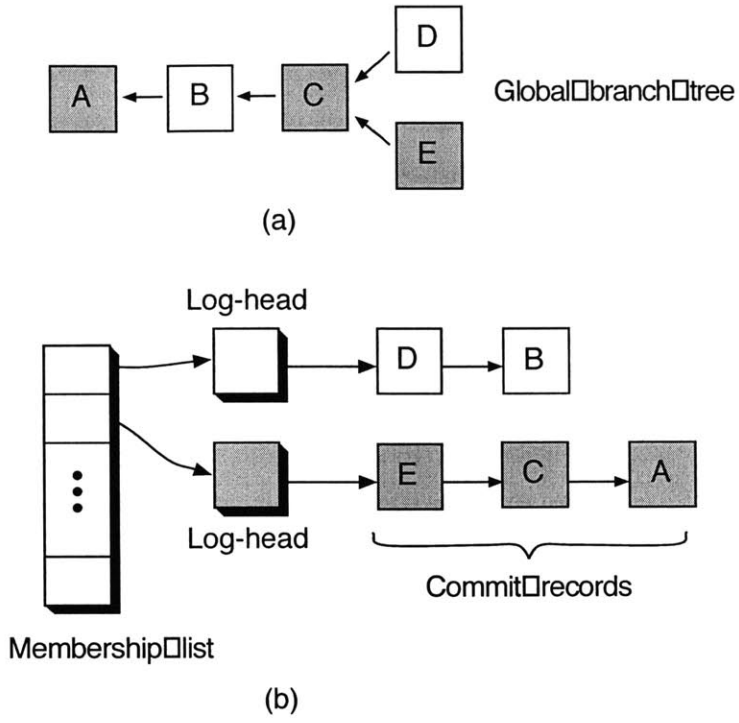

Figure 3-1: Pastwatch represents the shared branch tree in the DHT as a set of logs. Boxes of different colors represent commit records created by different users. Each commit record from the branch tree appears in a log. Each log contains all the commit records created by a single user. A membership list contains pointers to all the logs. Each log has a mutable log-head, which records the most recent commit record in the log. Each commit record also points to the previous, earlier commit record in the log. Mutable blocks are shown with shadows. All other blocks are immutable.

the tree. Using shared mutable blocks is not ideal. First, potentially unreliable users that could crash or disconnect makes locking an unattractive approach to serializing users' writes. Second, wait-free serialization is not possible with the simple `get/put` primitives that DHTs provide [18].

Pastwatch uses per-user logging to solve this problem. It represents the shared branch tree in the DHT as a set of logs, one log per user. Figure 3-1 shows the structure of a DHT branch tree and the corresponding logs. Each log contains a user's commit records. Pastwatch stores each commit record in an immutable content-hash block. It stores the GID of the most recent commit record in each log in a mutable public-key block, called the *log-head*. The log-head is signed with a user's private-key, so that only that user can modify the log-head. Each commit record contains the GID of the previous, earlier commit record in the log. For each user, Pastwatch uses a log, rather than

just a single mutable block that contains the user's latest commit record, because each user could have created the leaves of multiple branches on the DHT tree.

The project's membership list is stored as a mutable block. The membership list contains all the users' names and their log-head GIDs. Given the GID of the membership list, a user can fetch the membership list and scan the logs of all the users. The project administrator creates this membership list and inserts the block into the DHT, as part of the implementation of `createProject`.

The logs help users find all the commit records of a project. Per-user logging also isolates users' concurrent changes. If two users each append a commit record in two different partitions, their commit records appear on separate logs. In this case, the set of logs in the DHT remains consistent. After the partition heals, a user obtains the partitioned changes simply by scanning the latest versions of all the logs. The per-user logging technique was first developed for the Ivy peer-to-peer file system [29]. Ivy and Pastwatch use the technique to build different systems, however.

3.3.3 Operations on the DHT Branch Tree

The following two procedures access/modify the DHT branch tree.

- `fetchDHT(projID)` returns the DHT branch tree. It constructs a branch tree in the following manner.

    1. Use `projID`, fetch the membership list and downloads all the users' logs from the DHT. Store a log-head block on the user's computer if the local copy is older (i.e. by comparing version numbers of the two copies of the public-key block).
    2. Retrieve commit records from each log. For each commit record, also retrieve its parent and commit records it supersedes.
    3. Build a branch tree. Given a commit record `r`, use `r` in the branch tree if Pastwatch was able to retrieve `r`'s parent and commit records that `r` supersedes. In this case, also store `r` on the user's computer.

`fetchDHT` always return a consistent tree; it does not add a commit record to the tree if it cannot retrieve the commit record's parent or superseded nodes from the DHT.

- `extendDHT(projID,user,name,parent,supersede,contents)` adds a new commit record to the DHT branch tree.

    1. Create a new commit record using the arguments. Include the GID of the most recent commit record in the user's log (from the user's log-head) in the new commit record.
    2. Insert the new commit record into the DHT. Also store a copy locally.
    3. Update the user's log-head to include the GID of the new commit record. Store a copy of the new log-head on the user's computer. Return the new commit record.

`extendDHT` is deterministic: the identifier of the commit record corresponds to its contents, which are the call's arguments. Also, user `u1` cannot call `extendDHT(u2,...)` because `u1` cannot correctly sign `u2`'s log-head block. Section 3.5 discusses how Pastwatch uses `extendDHT`.

3.4 Local Branch Tree

Pastwatch stores a local branch tree's versions on the user's disk. Each version has an unique *version identifier* that Pastwatch can use to retrieve the version from disk. Each user's computer also keeps the following data.

- Mappings between version names and version identifiers.
- Mappings between commit record GIDs and version identifiers.
- Identifiers of leaf versions on the local branch tree.
- Local copies of log-heads and commit records retrieved/created by `fetchDHT` and `extendDHT`.

For each user `u`, the rest of this chapter uses `fetchLocal` in place of `fetchTree(u,proj)`, and `extendLocal(...)` in place of `extend(u,proj,...)`. `extendLocal` creates a new version, stores it on disk, updates the corresponding metadata, and returns the new version.

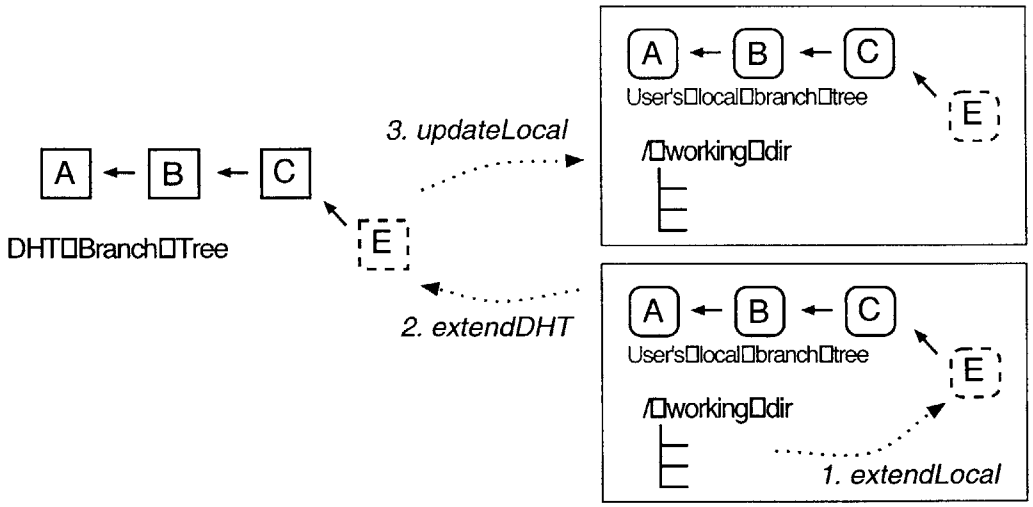

Figure 3-2: Interactions between a DHT branch tree (left) and users' local branch trees (right). A dashed box is the result of an operation represented by the nearby dotted arrow. A commit operation first adds a new version, E, to the user's local branch tree using `extendLocal`. It then adds E to the DHT branch tree using `extendDHT`. Finally, Pastwatch copies E fro the DHT branch tree into another user's local tree using `updateLocal`.

3.5 Synchronization

Pastwatch synchronizes a user's local branch tree against the DHT branch tree using most three operations, as shown in Figure 3-2. In this figure, a dashed box is the result of an operation represented by the nearby dotted arrow. First, when a user commits changes, Pastwatch adds a new version, E, to the user's local branch tree using `extendLocal`. Before the commit operation ends, it calls `extendDHT` to add E to the DHT branch tree as well. Pastwatch periodically adds new versions from the DHT branch tree to a user's local branch tree using `updateLocal`. For example, for the user in the top of the figure, Pastwatch adds version E to that user's branch tree. To increase the likelihood that a user can immediate see other users' new versions, Pastwatch also calls `updateLocal` before every user operation (e.g. check out).

Each user's local branch tree is always consistent: both `extendLocal` and `updateLocal` preserve this property. Furthermore, the DHT branch tree always appears to be consistent to users, because `fetchDHT` returns a consistent tree. Recall that in a consistent tree, each version only refers to (i.e. parent or supersede relation) versions that are in the tree.

3.5.1 Updating Local Branch Tree

A Pastwatch background process calls `updateLocal` periodically. `updateLocal` also runs before each user operation (e.g. checkout or commit). It works in the following manner.

1. Get a copy of the DHT branch tree, `T = fetchDHT(projID)`.
2. Walk `T` starting from the root. Sort all commit record into a partial order based on the parent and supersede relationship between them. That is, `r0<r1` if `r1` refers to to `r0` as a parent or supersedes `r0`.
3. Iterates the partial order, for each commit record `r`
   - Call `extendLocal(r.user,r.name,p,s,r.contents)` on the user's local tree, where `p`, `s` are `r.parent` and `r.supersedes` translated into version identifiers. Because `T` is consistent, this call should not fail.
   - Record mappings between the new commit record's GID and the new version's identifier on the user's computer.

At the end of `updateLocal`, the local tree contains every node in `T`. `T` may not equal to the local tree because the local tree could contain new versions yet to be added to the DHT, or, due to DHT inconsistency, the user could not retrieve some commit records that the user retrieved earlier.

`updateLocal` preserves the invariant that if a version exists on two users' local branch trees, then the version has the same user name, version name, parent, contents, and supersede list on both trees. `updateLocal` calls `extendLocal` with data from the same set of commit records to deterministically add the same new versions to each user's local branch tree.

3.5.2 Adding New Versions to the DHT

A commit operation uses `extendLocal` to add a new version to the user's local branch tree. It then adds the new version to the back of a persistent FIFO queue, stored on the local disk. Before the commit operation completes, Pastwatch tries to add each element in the queue to the DHT tree using `extendDHT`. It succeeds if it can contact the DHT. Otherwise, Pastwatch periodically tries to add elements from the queue to the DHT branch tree. If there are multiple new versions in the

queue, Pastwatch can batch the `extendDHT` calls so that it only needs to update the user's log-head once. For each new commit record, Pastwatch also records the mappings between the commit record GID and the version's identifier.

`extendDHT` may add a new commit record `r` to the DHT even if the user cannot retrieve a commit record `s` that is `r`'s parent or in `r.supersede`. This scenario arises if `s` is in the user's local tree, but because the DHT does not provide atomicity, the user could no longer retrieve `s`.

Because users may not always be able to update the DHT branch tree at the end of each commit operation, the DHT branch tree can be considered as a subtree of the union of all users' local branch trees. The DHT branch tree, therefore, does not contain all the states of a project.

3.5.3 Coping with Weak DHT Semantics

Because the DHT does not provide atomicity, a user may not be able to retrieve a block that the user or another user inserted or retrieved earlier. This means when Pastwatch calls `updateLocal`, `fetchDHT` may not return a tree that contains all the commit records that appear in all the logs; `fetchDHT` only uses a commit record if it can be reached from a log-head and that adding the commit record to the tree preserves the tree's consistency.

The implication is that after `updateLocal`, the user's local branch tree may not contain new versions that others have committed. If the window of inconsistency is short, then the next time the user issues an operation or when Pastwatch calls `updateLocal` again, it can download the new version into the local branch tree.

If a user can no longer retrieve a version that used to be in the DHT, Pastwatch re-inserts the version from the local tree. It periodically calls `updateDHT` to fix the DHT branch tree. `updateDHT` checks all the users' log-heads. If a DHT copy of a log-head is older (i.e. by comparing version numbers of the two copies of the public-key block) than the user's local copy, `updateDHT` re-inserts the log-head (already properly signed) and new commit records that the newer log-head refers to. Section 4.5 describes `updateDHT` in more detail. Pastwatch also reports GIDs of commit records that it cannot retrieve from the DHT. A user can always ask another user to re-insert those commit records explicitly.

Even if the DHT provides strong semantics, a user's local branch tree may still be stale; the user may be disconnected or a disconnected user has committed changes but has yet to re-connect to the

network. Pastwatch offers eventual consistency: if all users receive the same set of versions, they arrive at the same branch trees.

3.6 Optimization: Concurrency Control

If non-atomic behaviors are common in the DHT, then Pastwatch cannot update a user's local branch tree promptly. This means if users commit changes frequently, their new versions may form forks on their branch trees.

Fortunately, using techniques such as quorum consensus, a DHT may be able to provide atomicity in the common case. For example, when the following conditions all hold: static DHT membership, few host failures, no byzantine-faulty hosts, and no network partitioning among the DHT hosts. While a DHT system made of desktop computers of a large number of random Internet users may not provide this property, a provisioned DHT with a smaller number of hosts (e.g. tens to hundreds) may offer this property. Chapter 5 describes such a DHT.

Pastwatch tries to serialize users' commit operations to avoid forking. For connected users, it succeeds when the DHT provides write-to-read consistency. Pastwatch uses a variant of the Burns mutual exclusion algorithm [6]. It makes the algorithm non-blocking by imposing a lease on each user's lock. It does not use wait-free serialization because wait-free objects cannot be constructed using the primitives that a DHT offers [18].

Each user's log-head contains a `lock` variable. Acquiring a lock has three steps.

1. Check if anyone else's `lock` variable is non-zero. If yes, then someone else is trying to acquire a lock. Fail.

2. Otherwise, set the user's `lock` variable to non-zero.

3. Again, check if anyone else's `lock` variable is non-zero. If yes, then someone else is trying to acquire a lock. Set the user's `lock` variable to zero and fail. Otherwise, succeed.

Figure 3-3 shows the pseudo-code for the algorithm. `acquire`'s correctness depends on the DHT providing write-read consistency. Below is an informal proof. Assume there are two users, x, and y, both calling `acquire`.

```
1  acquire (hash logs[], hash mylog, hash v) {
2    log-head H[], h;
3    bool locked = false;
4
5    for (int i = 0; i < logs.size (); i++) {
6      H[i] = DHT.get (logs[i]);   // fetch a log-head
7      if (logs[i] == mylog) h = H[i];
8      else if (H[i].lock != 0) locked = true;
9    }
10   if locked
11     return "no lock";
12
13   h.lock = v;
14   DHT.put (mylog, h);
15
16   for (int i = 0; i < logs.size (); i++) {
17     H[i] = DHT.get (logs[i]);
18     if (logs[i] == mylog) h = H[i];
19     else if (H[i].lock != 0) locked = true;
20   }
21   if locked {
22     h.lock = 0;
23     DHT.put (mylog, h);
24     return "no lock";
25   }
26
27   return "has lock";
28 }
```

Figure 3-3: Pastwatch uses this algorithm to serialize connected users' commit operations.

- If x's second set of `DHT.get` calls (line 17) occurs before y's `DHT.put` completes (line 14), then write-to-read consistency guarantees that y's calls to `DHT.get` returns x's log-head with the lock set.

- If x's second set of `DHT.get` calls occurs after y's `DHT.put` completes, then one of those `DHT.get` returns y's log-head with the lock set.

When multiple users' Pastwatch software run `acquire` concurrently, it is possible that both calls to `acquire` fail. When a call to `acquire` fails, the Pastwatch software backs-off a random amount of time (e.g. a few seconds), then retries. If it cannot acquire a lock after T seconds (e.g. 2 minutes), Pastwatch proceeds to set the user's `lock` variable and continues as though it has obtained the lock. This situation occurs when a user holding the lock has crashed or left the partition.

3.7 Security Properties

Pastwatch has two security related design goals. First, it should allow users to recover from un-wanted changes committed by malicious users and intruders pretending to be users. Second, it should cope with a DHT's malicious behaviors. This section describes how Pastwatch accomplishes these goals from the perspective of the branch tree in the DHT and users' local trees.

3.7.1 Recover from un-wanted changes

A malicious user can only affect another user's local branch tree by adding new commit records to the branch tree in the DHT. A user can always recover from un-wanted changes by using files from a bad version's parent. A commit record can refer to commit records that do not exist in the DHT. `updateLocal` ignores these inconsistencies because `fetchDHT` always returns a consistent branch tree.

A bad commit record can also propose a name that was already used for a different version. Making sure that version names are unique is important. For example, if a user remembers `v` as a version with a desirable set of files, Pastwatch should not return a different set of files when the user checks out `v`. If a commit record proposes a name that has already been used, Pastwatch switches to using version identifiers to refer to the versions with duplicate names. If a user refers to the name, Pastwatch outputs the version identifiers and asks the user to pick one.

A user `u1` can change `u1`'s log-head to point to a different log, with different commit records. The old commit records are not lost; they are still in the DHT. If another user adds a new commit record that depends on a commit record from the old log, then the old commit record re-appears because `fetchDHT` uses the parent or supersede pointers to retrieve commit records not in a log.

It is easy to detect that a user has changed the user's log, rather than merely adding new commit records to it. In this case, the project administrator can remove the malicious user.

3.7.2 Coping with malicious DHT behaviors

Using self-authenticating content-hash and public-key blocks limits a DHT to two possible bad behaviors. It could deny the existence of a commit record or it could return a stale copy of a user's log-head. In the latter case, a user's `fetchDHT` may return a stale copy of the DHT branch tree.

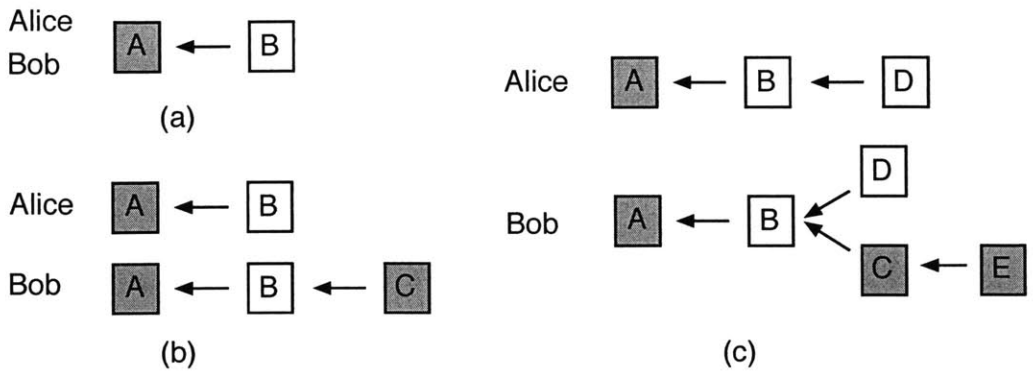

Figure 3-4: An example of a stale-data attack. The gray boxes are commit records created by Bob. The white boxes are those created by Alice. In (a), both users see the same DHT branch tree. In (b), the DHT hides Bob's new commit record, C, from Alice. For the attack to persist, the DHT must also hide any subsequent commit record that depends on C. This means Alice can no longer see any of Bob's new changes, such as E, even though Bob may be able to see Alice's. In this case, any out-of-band communication between Alice and Bob reveals the attack.

If a user cannot retrieve a DHT block `b` whose GID is in a log-head or commit record, and other users can produce local copies of `b`, then the DHT is misbehaving or experiencing inconsistency. If re-inserting `b` into the DHT does not solve the problem, users may want to switch to a new DHT.

Pastwatch uses a branch tree to detect inconsistencies due to a DHT hiding users' changes. Figure 3-4 shows an example. In the figure, gray boxes represent commit records created by Bob and white boxes represent commit records created by Alice.

- In part (a), both users see the same DHT branch tree.

- In part (b), Bob had created a new commit record C, but the DHT hides C from Alice. The attack has no effect if it stops before Alice commits new changes; the commit operation would ask Alice to update her working directory with changes from C if it sees C.

- Suppose the attack persists and Alice commits a new version D without seeing C. If the attack stops now, Alice would see a fork on her branch tree because both C and D use B as the parent. This tells Alice that the DHT has behaved inconsistently.

  If the attack persists, then the DHT must continue to hide all commit records that are descendants of C. For example, in part (c), the DHT must hide E from Alice as well. This means

Alice can never see any of Bob's new changes. Any out-of-band communication between them would discover this attack (e.g. sending an e-mail after each commit). This property is similar to fork-consistency [28], first introduced by Mazières and Shasha.

A network partitioning among the DHT hosts also gives the appearance of a stale-data attack. By itself, Pastwatch cannot distinguish a short-term attack from a network partitioning; in any case, if the DHT behaves in an inconsistent manner often, users may want to switch to a new DHT with a new set of hosts.

Chapter 4

Pastwatch Implementation

Figure 4-1 illustrates the Pastwatch software structure. There are three major pieces. First, a set of DHT hosts store the shared branch tree that Pastwatch uses to communicate users' changes. A project could use a DHT that consists of some of the project members' computers, or a public DHT [20]. For example, an open-source software project could use a DHT that consists of servers already donating resources to help distribute open-source software.

The other two programs run on users' computers. `past` is the Pastwatch client. Users run `past` to access their local branch trees. `past` runs `updateLocal` before every user operation, and `extendLocal` and `extendDHT` at the end of a commit operation. `upd` runs `updateLocal` periodically, and `updateDHT` and `extendDHT` if needed. Each user has an instance of `upd` running in the background.

Each user needs to know how to contact at least one DHT host, and preferably more in case that host is not available. Both `past` and `upd` establish TCP connections to one or more DHT hosts to run `DHT.put` and `DHT.get`. Each program picks the first established connection to use and closes all the other ones.

Figure 4-2 shows the major data structures in Pastwatch. On the left is a working directory. It contains a pointer to a versio on the local branch tree; this is the working version. Pastwatch stores each version as a *snapshot* object, as shown in the middle of the figure. To be efficient, each commit record in the DHT logs contains the set of deltas that, when applied to an existing snapshot, produces a new version's snapshot. Pastwatch remembers which commit record corresponds to which snapshot.

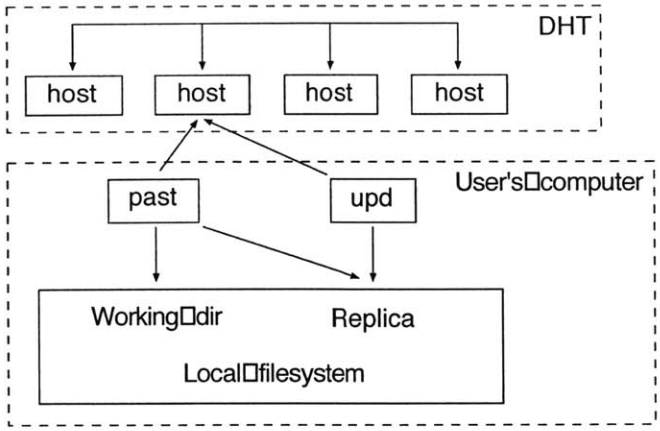

Figure 4-1: Pastwatch implementation. A dotted box represents a computer. `past` and `upd` programs run on users' computers. Users run `past` to issue user operations. `past` runs `updateLocal` before every user operation, and `extendLocal` and `extendDHT` at the end of a commit operation. `upd` runs `updateLocal`, `updateDHT`, and `extendDHT` in the background.

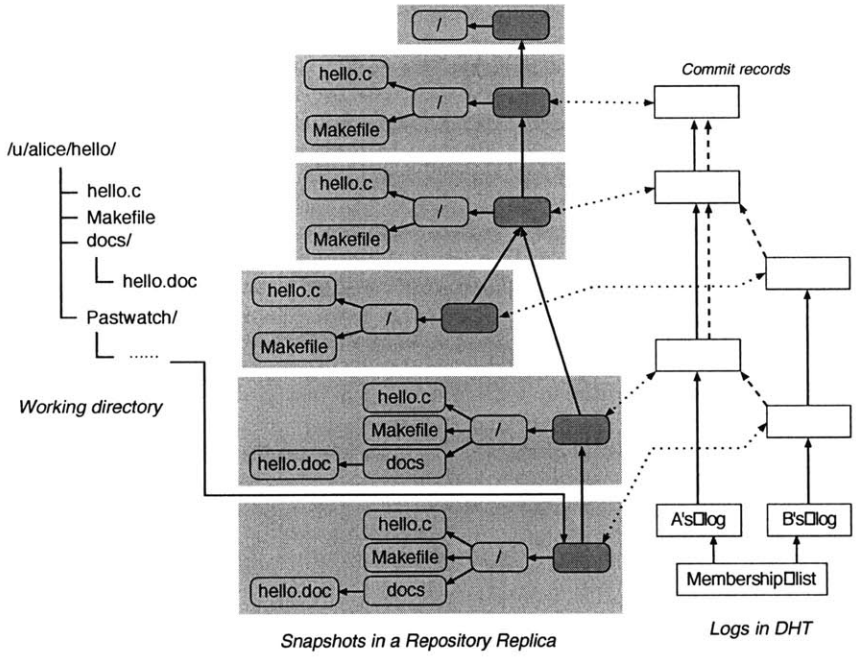

Figure 4-2: Pastwatch's data structures. Oval boxes are blocks stored on a user's computer. Each shaded region is a snapshot in a user's local branch tree. Each snapshot points to the parent on the branch tree (solid arrow). Rectangular boxes are the commit records, log-heads, and membership list in the DHT. Each commit record points to the previous commit record in the log (solid arrow), as well as a branch tree parent (dashed arrow). Pastwatch separately stores the mappings shown as the dotted arrows on the user's computer.

The Pastwatch software is written in C++ and currently runs on Linux, FreeBSD and MacOS X. It uses the SFS tool-kit [27] for event-driven programming and RPC. It uses the GNU diff and patch libraries for comparing different versions of a file and performing three-way merges. It uses the Rabin cryptosystem [46], with 1024 bit keys, to sign log-head blocks and verify signatures.

4.1 Local Storage

A user's computer stores blocks and some metadata for Pastwatch. Each block has an identifier k and a value v. Users access these blocks using `get(`k`)` and `put(`k`,`v`)`. Each identifier is an unique 160-bit value. This thesis refers to the identifier of a block stored on the local disk as *LID*. Just like a DHT, each host stores both content-hash and public-key blocks. Because both the DHT and the local storage use 160-bit values as identifiers, when Pastwatch stores a copy of a DHT block locally, the block's local LID is the same as its GID. On each user's computer, Pastwatch stores blocks in a SleepyCat[4] database.

Upon fetching a block, Pastwatch verifies that the LID matches the block's contents. This naming scheme allows different snapshots to share blocks; if two snapshots differ only at a few places, most of their blocks are shared. Pastwatch also uses the LID to detect illegitimate modifications to the blocks. This means users could send their local store to each other if necessary.

To improve performance, `past` and `upd` write data into the DHT in parallel and asynchronously whenever they can.

4.2 Snapshots

Each snapshot is a set of blocks, stored on the user's local disk, that form a directory hierarchy. It summarizes a version of the shared files at a point in time. The structure of this directory hierarchy is similar to that of a directory in a UNIX file system. See Figure 4-3. Pastwatch identifies files and directories using 160-bit file handles. Each file handle has an *i-node block* in each snapshot that stores a file or directory's attributes and pointers to its contents. For example, a directory i-node contains names of the directory entries and pointers to these entries' i-node blocks.

Each snapshot uses a *file map* to record the LID of the i-node block for each file handle. This level of indirection is necessary because each block is immutable, so to change an i-node block,

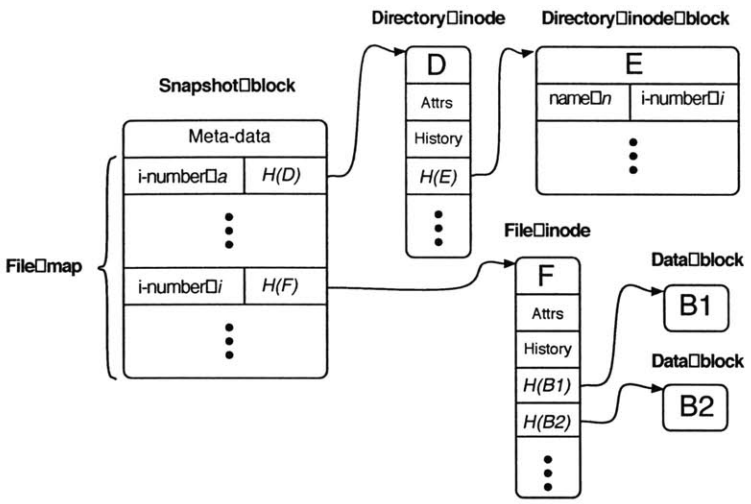

Figure 4-3: Each snapshot captures a version of the shared files as a directory hierarchy. $H(A)$ is the LID (i.e. the SHA-1 cryptographic hash) of A.

Pastwatch actually produces a new i-node block, with a new LID. The new snapshot's file map block, therefore, must correctly map the file handle to the new i-node's LID. Pastwatch stores the file map in a meta-data block. The LID of the meta-data block is the *snapshot identifier*, or *SnID*. Each meta-data block also contains the SnID of the parent snapshot. Using these parent pointers, Pastwatch can compute the parent and children of any version.

Initially, for each user, the state of a new project without any committed changes contains one snapshot. This snapshot contains a single i-node that represents an empty *root directory*. When a user commits a set of changes to an existing snapshot, Pastwatch makes a copy of the existing snapshot, replays the changes to the new copy, updates each file handle's corresponding i-node LID in the file map, and finally stores all modified blocks as new blocks.

Figure 4-4 shows an example. Each oval box in the figure represents a block, whose LID is shown in italic on top of the box. Non-italic hex values are file handles. For simplicity, all LIDs and file handles are abbreviated. The set of blocks in the top portion of the figure represents a snapshot. The corresponding directory hierarchy is shown on the right. The root directory has file handle `c9kH` and contains one entry with file handle `n6hm`. The file map has LID *nWS2* – this is the SnID of the snapshot.

The middle of the figure shows three changes that a user makes to files from the first snapshot.

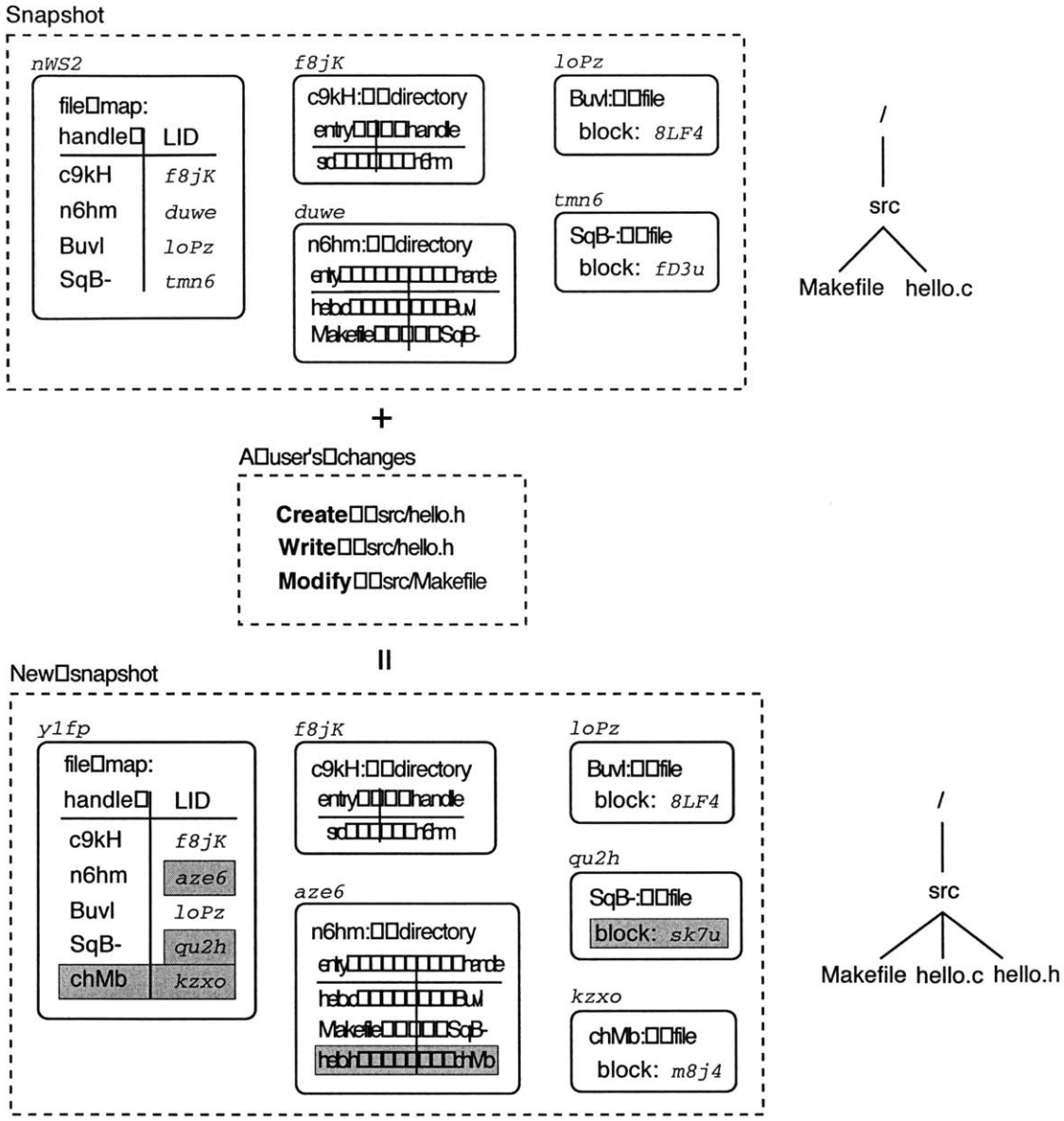

Figure 4-4: Constructing a new snapshot. Each oval box is a block whose LID is shown in italic on top of the box. Non-italic hex values represent file handles. All LIDs and file handles are abbreviated. The root directory has file handle `c9kH`. The SnID of a snapshot is the LID of the block that stores the file map. In this example, applying three changes to snapshot `nWS2` resulted in a new snapshot `y1fp`. Each shaded region represents a change to the corresponding i-node's block in `nWS2`. Each block is immutable; any change to a block's content results in a new block with a new LID. On the right of each snapshot is the corresponding directory hierarchy.

These changes create a new file, write some data to this file, and modify the contents of another file. When the user commits these changes, Pastwatch records these changes in a commit record. Section 4.3 describes the format of each commit record.

The bottom portion of the figure represents the new snapshot that results from applying these changes to *nWS2*. The shaded region highlights the changes. Note that each block with a change now has a different LID. For `Create`, Pastwatch creates a new i-node block for file handle `chMb` and inserts a new directory entry into the i-node of `n6hm`, changing the LID of that i-node to *aze6*. For `Write`, Pastwatch inserts the LIDs of new file contents into the i-node of `chMb`. As a result, `chMB` now has a new i-node block, with LID *kzxo*. For `Modify`, Pastwatch modifies the i-node of `SqB-` to point to new contents. Finally, Pastwatch modifies the file-map to reflect the new i-node LIDs. The new file-map is stored in a block with LID *y1fp* – this is the SnID of the new snapshot.

Pastwatch has a level of indirection between file names and file handles, so that the version history of a file is preserved across renames. Each file handle also has a *revision number*, initially set to one, and stored in the file handle's i-node block in each snapshot. A file handle has a higher revision number than that of the same file handle in a previous snapshot if the new snapshot contains new changes to the file. This implies each branch on the user's branch tree has a separate revision history. Each i-node block contains this revision history. This history contains the SnIDs of a set of snapshots that contain earlier revisions of the file. The k^{th} entry on the revision list points to a snapshot that contains the file at 2^{k-1} versions ago. This means searching a list takes $O(\log N)$ time, where N is the total number of revisions to the file. Revision numbers allow users to easily retrieve an ancestral version of a particular file or directory.

4.3 Structure of a Commit Record

Each commit record in a user's log corresponds to a version, and thus a snapshot. Instead of storing the entire state of a snapshot in a commit record, each commit record contains a set of changes that, when applied to the parent commit record's snapshot, produces the new snapshot.

Figure 4-5 describes the structure of a commit record. `writer` is the GID of the log-head of the user who created the commit record. `seqn` is a per-user sequence number. The first commit record Pastwatch appends for a user has a sequence number of 0. Each additional commit record

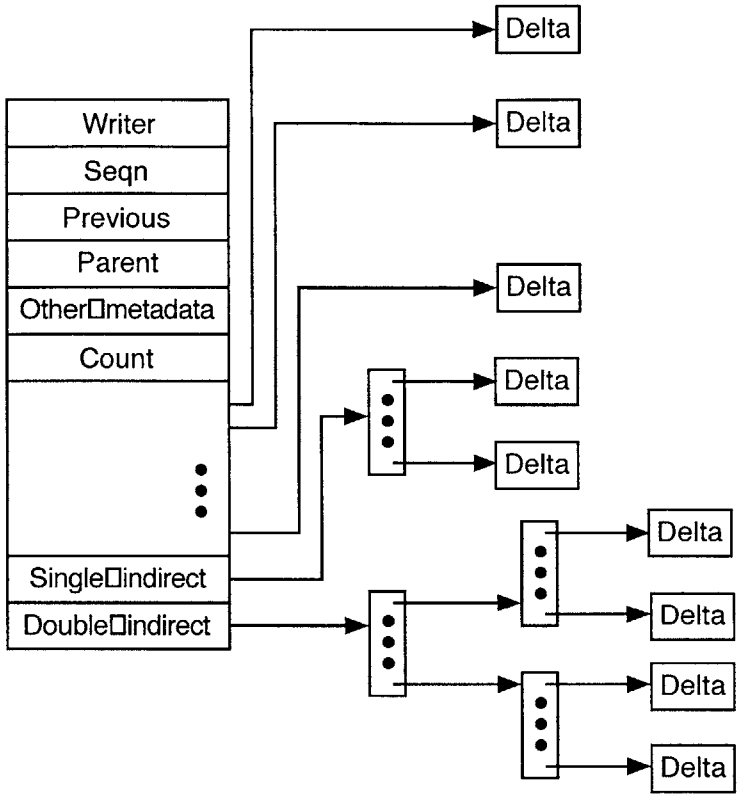

Figure 4-5: Structure of a Commit record is shown on the left. Each commit record points to a set of deltas, shown on the right.

increments the user's sequence number by 1. `previous` specifies the GID of the previous commit record in the log. `parent` specifies the GID of the parent commit record.

Each commit record records some changes to the project's directory hierarchy of shared files. The `count` field in a commit record specifies the total number of changes listed in the commit record. Each change is stored in a separate content-hash block called a *delta*. Pastwatch stores GIDs to the deltas in the commit record and using single and double indirect blocks, much like how a UNIX file system keeps track the addresses of a file's blocks.

Table 4.1 describes the types of deltas that Pastwatch records. Pastwatch identifies files and directories using 160-bit file handles. A delta describes a single modification to a file or a directory and contains the file handle of that file or directory. Users choose file handles randomly to minimize the probability of picking the same file handle for different files.

Type	Fields	Function
`Create`	type (file, directory, or symlink), file handle, mode	create new i-node
`Content`	file handle, GIDs of data blocks	describe new contents of a file
`Patch`	file handle, GIDs of patch contents	describe a patch to a file
`Link`	file handle, file handle of directory, name	create a directory entry
`Unlink`	file handle of directory, name	remove a file from a directory
`Rename`	file handle of old and new directory, old and new name	rename a file
`Tag`	tag (string), name of the version	create a repository tag

Table 4.1: Summary of the types of deltas in a commit record.

4.3.1 `Write` Delta

A `Write` delta contains GIDs of data blocks that make up the contents of a file. After Pastwatch fetches a `Write` delta, it also fetches each of the data block and store them on the user's computer. The LIDs of the data blocks in local storage are the same as their GIDs. For each regular file (i.e. not a directory), the file's i-node in a snapshot contains LIDs to data blocks as well. Therefore, a snapshot and a `Write` delta may share data blocks.

4.4 `updateLocal`

`past` runs `updateLocal` before each user operation to bring the user's local branch tree up-to-date. `upd` also runs `updateLocal` periodically. Because only this is the only place that calls fetchDHT, `updateLocal`'s implementation includes `fetchDHT`. Figure 4-6 describes this implementation.

There are three parts to this implementation. In the first `foreach` loop, Pastwatch fetches all the new commit records (i.e. it has not build snapshots for them before) reachable by following the `prev` pointers in all the logs. From lines 26 to 41, Pastwatch fetches all the commit records that it does not know about, but are dependents of commit record it fetched earlier. These "missing" commit records could come from the logs of users who are no longer part of the project, or Pastwatch might have fetched a stale log-head that does not point to some recent commit records. Finally, from lines 42 to 51, Pastwatch builds new snapshots.

`updateLocal` maintains the consistency of the local branch tree: it only builds a new snapshot if that snapshot's parent and superseded snapshots exist locally (lines 32 and 40 verify this).

Pastwatch cannot use a commit record to build a snapshot if the commit record contains incon-

```
1  hashtable<hash,snapshot> c2s; // in replica: maps commit record GID to snapshot
2  bool updateLocal (hash projID) {  // assumes can contact the DHT
3    bool fix = false;
4    commit_record t[];
5    membership_list m = DHT.get (projID);
6    foreach hash l in m.logs {
7      log-head h_dht = DHT.get (l);
8      log-head h_rep = replica.get (l);
9      if (h_dht == nil) continue;
10     if (h_rep == nil or h_dht.version > h_rep.version)
11       replica.put (l, h_dht);
12     else if (h_dht.version < h_rep.version) fix = true;
13     // fetch new commit records from log
14     hash k = h_dht.prev;
15     while ((k notin c2s.dom ()) and (k notin t) and k != 0) {
16       commit_record c = DHT.get (k);
17       if (c) { t.push_back (c); k = c.prev; }
18       else k = 0;
19     }
20   }
21   bool changed = true;
22   while (changed) {
23     changed = false;
24     bool ok = true;
25     foreach commit_record c in t {
26       if ((c.parent notin c2s.dom ()) and (c.parent notin t)) {
27         changed = true;
28         commit_record n = DHT.get (c.parent);
29         if (n) t.push_back (n);
30         else { t.remove (c); continue; }
31       }
32       if (c.parent notin c2s.dom ()) ok = false;
33       foreach hash s in c.supersede {
34         if ((s notin c2s.dom ()) and (s notin t)) {
35           changed = true;
36           commit_record n = DHT.get (s);
37           if (n) t.push_back (n);
38           else { t.remove (c); continue; }
39         }
40         if (s notin c2s.dom) ok = false;
41       }
42       if (ok) {
43         changed = true;
44         snapshot n = build_new_snapshot(c2s(c.parent), c);
45         if (n) {
46           c2s.insert (c.gid (), n);
47           replica.put (c.gid (), c);
48           update replica's metadata
49         }
50         t.remove (c);
51       }
52     }
53   }
54   return fix;
55 }
```

Figure 4-6: Pastwatch's implementation of `updateLocal`.

```
1  updateDHT (hash projID, bool push = false) {
2    hash to_update[];
3    membership_list m = DHT.get (projID);
4    foreach hash l in m.logs {
5      log-head h_dht = DHT.get (l);
6      log-head h_rep = replica.get (l);
7      if (h_rep and h_dht == nil or h_rep.version > h_dht.version) {
8        DHT.put (l, h_rep);
9        hash stop = 0;
10       if (h_dht != nil) stop = h_dht.prev;
11       for (hash k = h_rep.prev; k != stop; k++) {
12         commit_record c_rep = replica.get (k);
13         if (DHT.get (k) == nil) DHT.insert (k, c_rep);
14         k = c_rep.prev;
15       }
16     }
17   }
18   if (!push) return;
19   foreach snapshot x in replica's list of leaf versions {
20     commit_record c = replica.get (x's commit record's GID);
21     hash k = c.gid ();
22     while (k) {
23       if (DHT.get (k) == nil) DHT.put (k, c);
24       for (int j=0; j<c.deltas.size(); j++)
25         if (DHT.get (c.deltas[j]) == nil)
26           DHT.put (c.deltas[j], replicas.get (c.deltas[j]));
27       k = c.parent;
28       if (k != 0) c = replica.get (k);
29     }
30   }
31 }
```

Figure 4-7: Pastwatch's implementation of `updateDHT`.

sistent deltas that cannot be applied to the parent snapshot (line 44 and 45). An example is when the parent snapshot contains a file "foo/bar", and the new commit record contains a `Link` delta that also creates "foo/bar". In this case, the commit record and all other commit records that depend on it are ignored. This scenario could only happen if a user has maliciously created an inconsistent commit record. In this case the project administrator could remove the user's log from the membership list.

4.5 `updateDHT`

`updateLocal` returns TRUE if the user's local copy of a log-head is newer than the copy in the DHT. In this case, `upd` runs `updateDHT` to "fix" the branch tree in the DHT. Figure 4-7 describes Pastwatch's implementation of `updateDHT`.

`upd` calls `updateDHT` with the `push` variable set to false. In this mode, `upd` detects and recovers from the DHT serving a stale copy of a log-head. This scenario could arise, for example, when a network partitions or if a mobile user moves from one partition into another. In this mode, `updateDHT` does not check if every version in a user's local tree exists in the DHT.

If some users complain that one or more commit record cannot be fetched from the DHT even after `updateDHT` runs, a user could explicitly insert those commit records or run `updateDHT` with the `push` variable set to true. In the latter case, `updateDHT` exhaustively checks every version on the local tree and re-inserts local copies of all the commit records that are missing in the DHT. A user could also run `updateDHT` this way to insert a project's branch tree into a new DHT.

4.6 Users and Names

Table 4.2 summarizes the different types of names used by Pastwatch.

4.6.1 Project Members and Administrator

A project's administrator creates, manages, and signs the membership list in the DHT. Each user's name must be unique. One way to assign unique names is to use users' e-mail addresses, such as `alice@mit`. The Pastwatch software stops to work if two users have the same name. In this case a user can complain to the project administrator.

The project administrator can remove a user from the membership list. Pastwatch automatically assigns and uses an unique 160-bit user name – the user's log-head GID – to refer to that user's committed versions. This means Pastwatch does not need to keep a list of removed users.

Project members could decide to switch to a new administrator. The new administrator needs to create and sign a new membership list and insert that list into the DHT. The GID of the new membership list is the new project ID. The new membership list could contain the same set of logs as the old list; in this case the new project inherits the DHT branch tree of the old project. Each user also needs to tell Pastwatch to start using the new project ID.

Types of name	Example
Project name	`mfNqkAMnOXsSGCvgPYV3XlixzqM`
User name	`alice@mit`
Version name	`alice@mit:2`
Version of a file	`alice@mit:2`
Symbolic tag	`alice@mit:thesis_draft`
Branch tag	`alice@mit:mainb`

Table 4.2: Different types of names used by Pastwatch.

4.6.2 Version Name

Pastwatch names each version using a user name followed by a sequence number. For example, `alice@mit:2`. A user can retrieve any version of the files using its name. Pastwatch guarantees that no two versions have the same name, and that a name given to a version cannot be re-bound to a different version.

Some version control systems use names that capture branch information and revision history. For example, CVS [4] uses version names such as `1.546`, or `1.1.1.230`. These names embed the structure of the branch tree. For example, version `1.546` is a later version than `1.545`, and version `1.1.1.230` is on a different branch than `1.1.2.229`. Pastwatch names do not embed branch history. Instead, given a version name, Pastwatch can output the version's parent and children.

Internally, Pastwatch uses snapshot identifiers (SnIDs) to name different versions. It maintains mappings between SnIDs and human-readable names on the user's computer. When a user uses a name, such as `alice@mit:5`, Pastwatch translates the name into a SnID. Internal data structures (e.g. the working directory's working version and supersede list) refer to versions using SnIDs. When providing feedbacks to users, Pastwatch translates SnIDs back to human-readable names.

Using SnIDs internally makes switching version names easy. A name switch happens when Pastwatch discovers two commit records proposing the same `user:seqn` names. In this case, Pastwatch switches to using SnIDs as version names.

4.6.3 File/Directory Version or Revision

A user can refer to a particular revision of a file or directory using either a revision number (see Section 4.2) or the name of the version that produced the revision. For example, if version `al-`

`ice@mit:2` first introduced the file `foo`, then a user modifies `foo` in a new version `bob@mit:2`, then the first two revisions of `foo` are `alice@mit:2` and `bob@mit:2`.

4.6.4 Symbolic Tags

A user may want to associate a version of all the files with a name that is more meaningful to the user. Pastwatch allows a user to tag a version with an arbitrary tag. Each *symbolic tag* is the name the user applying the tag followed by an ASCII string. For example, `alice@mit:thesis_draft`. Each user's computer stores a mapping from symbolic tags to version names.

Pastwatch does not allow a single user to assign the same symbolic tag to two different versions. Furthermore, because the user's name is part of a symbolic tag, two users can never assign the same symbolic tag to two different versions of the shared files.

4.6.5 Branch Tags

When committing changes, the user can specify a branch tag for the new version. All versions that are descendants of this version inherit the branch tag, unless a new branch tag is given. When a user gives Pastwatch a branch tag, Pastwatch translates it into the name of the leaf version on that tree branch. A branch tag has the format as a symbolic tag.

When updating a working directory, Pastwatch tries to follow a tree branch with the same branch tag as the current working version of the working directory. Section 4.7 describes this in more detail.

4.7 Working Directory and User Operations

Table 4.3 summarizes common Pastwatch user operations. Section 2.3 describes some of these operations as well.

The *import* operation copies new files from the local computer into the project. A user specifies the name of a new directory to create in the project's directory tree of shared files, as well as a set of files on the user's local computer. Pastwatch creates a new version that contains the new directory, with the files from the user's local computer as its entries.

The *checkout* operation creates a new working directory that contains files and directories from a directory in a version on the user's local branch tree (i.e. the working version). For simplicity, this

Command	Function
`latest`	Reports names of leave versions
`import`	Creates files and directories in the project
`checkout`	Copies files and directories into working directory
`update`	Updates working directory against a version
`merge`	Merges changes from another branch into working directory
`commit`	Commits changes from the working directory
`rename`	Renames a file in the project and working directory
`tag`	Creates a symbolic tag that maps to a specific version
`branch`	Creates a new branch on the branch tree
`diff`	Compares different versions of a file or directory
`add/remove`	Adds/removes a file/directory on the next commit

Table 4.3: The most common Pastwatch commands.

Variable	Description
`info.project`	Project name
`info.keyfile`	User's private-key, stored on the user's computer
`info.btag`	Branch tag
`info.version`	The working version
`info.supersede`	The supersede list

Table 4.4: Important variables stored in `Pastwatch/info`.

thesis refers to each directory or sub-directory in the checked-out version of the directory hierarchy as a working directory. For example,

```
bob> past -p mfNqkAMnOXsSGCvgPYV3XlixzqM checkout src
D src
U src/Makefile
U src/hello.c
bob> cd src
bob> ls
Pastwatch/    Makefile    hello.c
```

`checkout` creates a `Pastwatch` directory in each working directory. This directory contains the working directory's metadata. Table 4.4 summarizes these variables. In Figure 4-2, the arrow between the working directory and the bottom snapshot is the value of `info.version`.

4.7.1 Updating a Working Directory

The *update* operation merges new changes from a version, called the *update source*, into the working directory. By default, the update source is the leaf version that is a descendant of the working directory's current version. For example,

```
bob> past update                    % getting changes from new leaf
pastwatch: updating .
U hello.h, new file
P hello.c
```

A user could also specify an update source explicitly, using a version name or symbolic tag. Pastwatch uses the UNIX `diff3` utility to implement part of the merge process.

4.7.2 Per-file Version Control

Pastwatch can update a file to a particular version. For example,

```
bob> past update -t alice@mit:3 hello.c
pastwatch: updating hello.c
Merging changes into hello.c
P hello.c
```

In this example, Pastwatch downloads an earlier version of `hello.c` into the working directory. Afterward, it records the fact that `hello.c` is of a different version than the rest of the working directory in `Pastwatch/db`, a metadata file. Subsequent update operations no longer affects `hello.c` (i.e. no new changes will be merged into this file) until the user explicitly updates `hello.c` to the latest version again. For example,

```
bob> past update
pastwatch: updating .
U hello.h, new file
warning: hello.c is from a different version
bob> past update -A
pastwatch: updating .
P hello.c
```

4.7.3 Add, Remove, Rename Files and Directories

Pastwatch allows users to *add*, *remove*, and *rename* files and directories. `rename` takes effect immediately: after verifying that the working version is a leaf version, Pastwatch creates a new version that captures the effect of the rename operation and adds the new version as the new leaf.

A user can add and remove files and directories from the project's shared files. `add` and `remove` register these *directory operations* in `Pastwatch/ops`. The next time the user calls `commit`, Pastwatch inserts these operations as `Create`, `Link`, and `Unlink` deltas in the new commit record. The `add` command also allows a user to re-link a file that was removed earlier.

A directory operation registered in `Pastwatch/ops` may conflict with another user's committed changes. `update` reports these conflicts.

4.7.4 Tagging

The tag operation attaches a symbolic tag, of the format `user:string`, to a version of the files. To tag a version, Pastwatch appends a commit record that points to a `Tag` delta. The delta specifies the string portion of the tag and the GID of the commit record whose corresponding version the user wants to tag. For example, if a `Tag` delta specifies `thesis_draft` and `wJyB`, and the delta appears in a commit record with user `alice@mit`, then the delta defines a mapping from `alice@mit:thesis_draft` to `wJyB`'s corresponding version. Pastwatch records symbolic tag mappings on each user's computer.

4.7.5 Explicitly Creating a New Branch

A user could issue `branch` to explicitly create a branch. `branch` is similar to `commit` in that it creates a new version that contains the changes from the working directory. It differs from `commit` in that the working version does not need to be a leaf, and that it takes an additional branch tag argument. The new version uses `info.version` as the parent and `info.supersede` as the supersede list, and has the new branch tag. The commit record contains the new branch tag in a `Tag` delta. The new snapshot that corresponds to this version contains the new branch tag.

A user can also issue `branch` to commit changes from a working directory whose working version has been superseded.

When Pastwatch creates a working directory from a version of the shared files, it stores the version's branch tag in `info.btag`. After another user explicitly creates a new branch with a different tag, Pastwatch can use `info.btag` to following the original branch.

4.8 Managing Projects

To start a new project, a project administrator calls an utility program, `pastview`, to create a private/public-key pair for the project, a private/public-key pair for the user, a membership list, and a log-head.

```
alice> pastview -n hello
Creating new key: /home/alice/.pastwatch/hello/view.key
New project GID: mfNqkAMnOXsSGCvgPYV3XlixzqM
Creating new key: /home/alice/.pastwatch/hello/key.0331
New user log-head GID: BEk3vsTzEU+9zB53MbpXAAYGTNk
Move key to: /home/alice/.pastwatch/hello/key.BEk3vsTzEU+9zB53MbpXAAYGTNk
```

The name `hello` merely tells Pastwatch where to store the key files, it does not have any other meaning. In this example, `pastview` creates a membership list with GID `mfNq...`. The membership list contains one log-head, whose GID is `BEk3...`. The first member of the project owns the project's private-key, and, by convention, is the project's administrator. `pastview` directly contacts a DHT host: it inserts both the membership list and the new log-head into the DHT. It does not create a local branch tree for the user. `past` creates a new branch tree when it accesses a project for the first time (e.g. on a checkout operation).

Another user can call `pastview` to create a new private/public-key pair and a new log-head block and send the new log-head GID to the administrator. The administrator can then add the user to the membership list.

4.9 Checkpoints

When a user checks out or imports files into a project for the first time, Pastwatch creates a local branch tree for the user. It contacts the DHT to download all the logs and commit records for the project, and builds a snapshot for each commit record. If the project has existed for awhile, there may be many commit records to fetch and many snapshots to build.

Pastwatch offers two solutions for this problem. First, To make one-time checkouts more efficient, each user can periodically publish a *checkpoint* of a version of the shared files. A checkpoint is a single *checkpoint record* using the same format as a commit record. It contains deltas that, when applied to an empty snapshot, produce a snapshot whose contents are the same as the contents of a

version. Each user's log-head separately points to a single checkpoint record. `fetchDHT` ignores checkpoint records.

A project member can create a checkpoint by using the `makecp` operation. `makecp` takes a version as an argument, checks out that version into a working directory, creates a new checkpoint record using the files and directories in the working directory, inserts the checkpoint record and its delta blocks into the DHT, and finally updates the user's log-head.

Non-project members can use the `fetchcp` operation to fetch a checkpoint. The fetchcp operation takes the name of a user, fetches the checkpoint record from that user, builds a new snapshot, and creates a working directory from that snapshot. It stores the snapshot locally, but does not create a local branch tree. A user cannot commit changes from this working directory.

Alternatively, a user may download a package that contains a project member's full branch tree, the corresponding metadata, and DHT blocks. There could be a web server that stores and serves these packages for different projects. This is only an optimization; Pastwatch does not depend on the web server for correctness or availability.

When a user downloads another user's branch tree, the branch tree could be corrupt. Pastwatch leaves verification of the branch tree to external, out-of-band mechanisms. For example, after each commit, the user who committed changes could send an e-mail that describes which files have changed and the snapshot identifier of the new version. If a user's branch tree is wrong, then using the new commit record, Pastwatch would build a different snapshot with a different snapshot identifier. Note that users can easily verify the integrity of a snapshot from a downloaded branch tree because the snapshot consists of self-authenticating content-hash blocks.

Chapter 5

Cooperative Storage for Pastwatch

The feasibility of Pastwatch's design depends on the premise that a practical DHT can be built and deployed. This chapter describes the design of such a DHT, called *Aqua*. Aqua differs from existing DHTs [11, 12, 16, 26, 33, 38, 48] in that the focus of its design is on given DHT hosts control over their contributed resources. This feature makes Aqua practical; contributors are more willing to donate resources if they know what their resources will be used for.

Aqua is specially designed for Pastwatch. Pastwatch uses Aqua to implement each project's shared branch tree. In turn, Aqua exploits properties of Pastwatch to keep the system practical and simple. For example, each Aqua host uses Pastwatch as a tool to detect and remove data that it does not want to store. Also, Aqua's design is simple in that it only provides best-effort consistency guarantees – this works because Pastwatch, at the application level, handles inconsistencies that may arise from the DHT.

The design of Aqua is joint work with Alex Yip. Aqua's implementation is influenced by many existing DHT systems, particular DHash [11, 12]. Its development is an on-going work; this thesis describes a prototype implementation that is in-use today.

5.1 Managing Resources

A DHT stores data at the granularity of blocks. It spreads these blocks onto multiple hosts; each host only has a subset of all the data for an application. Without the application's help, each host cannot easily determine the purpose of each block. This complicates several administrative tasks.

For example, a host may not be able to prevent malicious users from using its resources to store un-wanted data. For legal, political, or personal reasons, this lack of control may not be acceptable.

Ideally, the owner of a host that contributes resources to Aqua can 1) check that the blocks the host stores belong to a Pastwatch project, 2) examine the content of each project, and 3) remove (i.e. garbage collect) blocks that belong to an un-wanted project. Aqua exploits the structure of Pastwatch to support these administrative tasks.

5.1.1 Removing Illegitimate Blocks

Each Aqua system has a central administrator. The administrator creates, signs, and distributes a *project list* that names all the projects that want to use the system. When a project decides to use an Aqua system, the project administrator asks the Aqua administrator to add the project to this list. The list contains the GID of each project's membership list.

Pastwatch offers a program, `p.audit`, that each host owner uses to determine the set of blocks that belong to a project. For each project, `p.audit` contacts the DHT, scans the project's logs, and reports the GIDs of all the DHT blocks that the project uses. These blocks include the commit records reachable from the logs in the project's membership list, the delta blocks that these commit records refer to, and commit records, from logs of past project members, that are still part of the project's DHT branch tree.

Aqua requires every block to contain the GID of a project's membership list. Using this GID, each host maintains a mapping between blocks and projects. If a host stores a block k for project p, but `p.audit` does not report k as part of project p's data, then the host can safely remove k.

A user can call `p.audit` incrementally. Because a commit operation inserts a new commit record at the front of an user's log, to determine the GIDs of new blocks inserted since the last time it ran, `p.audit` only needs to fetch users' log-heads and a few commit records. If a typical project has only a few commit operations per-day, then a daily audit of all the projects could be very fast.

5.1.2 Content Audit

A host's owner can use Pastwatch to check out versions of a project's shared files and directories. Based on the contents of the resulting working directory, the user can determine if the host should continue to store data for the project. For example, if a public Aqua system supports open-source

software, but a project contains many encrypted files, then some hosts' owners may choose not to store data for that project.

Examining the contents of all the projects may take a long time. In practice, examining projects with large storage requirements may adequately discourage abuse.

This approach does not prevent users from disguising illegitimate data. For example, some users may be able to encode pirated software or music as source-code files and trick hosts into storing these files.

5.1.3 Garbage Collection

To remove a project, a host's owner runs `p.audit` to determine the GIDs of all the blocks that the project uses and removes those blocks that the host stores. Because each block contains the GID of the project's membership list, projects do not share blocks among themselves.

Currently, if a host refuses to store data for a project, the Aqua administrator must remove the project from the project list so that other hosts do not replicate data for that project onto the host. Future work explores how to let hosts with different interests co-exist in one system.

5.2 Limiting DHT Membership

Each Aqua system limits which hosts can participate in the system. Limiting membership simplifies mapping a block onto a host. It also means there is a way to exclude hosts with inadequate resources, hosts that fail frequently, and misbehaving hosts.

Each Aqua system requires an administrator. The administrator controls which host can participant in the DHT. If a project uses an Aqua system composed of only project members' computers, then the Aqua administrator could be the same as the project's administrator. Otherwise, the administrator of a public Aqua system, shared by many projects, could be a user that actively solicits resources for the system.

Each Aqua administrator maintains a *host list* of computers that are willing to contribute resources to Aqua. Some additional admission criteria may be that a computer needs to have sufficient bandwidth and storage resources to be useful and that it has not been known to misbehave [7]. For example, a system may want to avoid hosts with low bandwidth (e.g. cable modem users) to opti-

mize performance, and avoid hosts with bad uptime to minimize replica maintenance overhead [5].

Operationally, the administrator signs and distributes the host list to all the computers on the list. Each version of the list contains an increasing sequence number. While performing DHT operations, hosts compare sequence numbers of their lists pairwise and retrieve new lists from each other. This means even if a host fails to receive a new list from the administrator, it can obtain the list in a peer-to-peer fashion soon afterward. Each entry in the list contains a host's IP address, port number, and a 160-bit host identifier. When a host contacts a running host to gain entry into the system, the running host verifies that the new host appears on the host list. A bad host is removed from the system when all hosts with pointers to the bad host have received a new host list.

5.3 Implementation

Each host runs a program, `aqua`, to store and serve blocks using a SleepyCat database. If `past` needs to fetch a block whose GID is k, it contacts an `aqua` server. The server locates the Aqua host that is responsible for storing k, sends an RPC to that host to fetch the block, and returns the result to `past`. `past` communicates with the client half of an `aqua` program via TCP. `aqua` programs communicate among each other using TCP or UDP. In the latter case, each program keeps estimated round-trip latencies to all the other hosts and retransmits potentially lost RPCs. All RPCs are idempotent.

5.3.1 Data Types

Aqua implements the DHT abstraction from Section 3.3.1.

5.3.2 Consistent Hashing

Aqua has an 160-bit address space. The GID of each block is an unique 160-bit value. Each computer also has an unique 160-bit host identifier. Aqua uses a variant of consistent hashing [19] to aggregate resources from multiple computers: for each block with GID k, it stores k on the R hosts whose identifiers are or immediately follow k. These hosts are k's *successors*. Aqua's address space is circular: the immediate successor of $2^{160} - 1$ is 0. In the absence of failure, given a set of hosts, each block always maps to the same hosts.

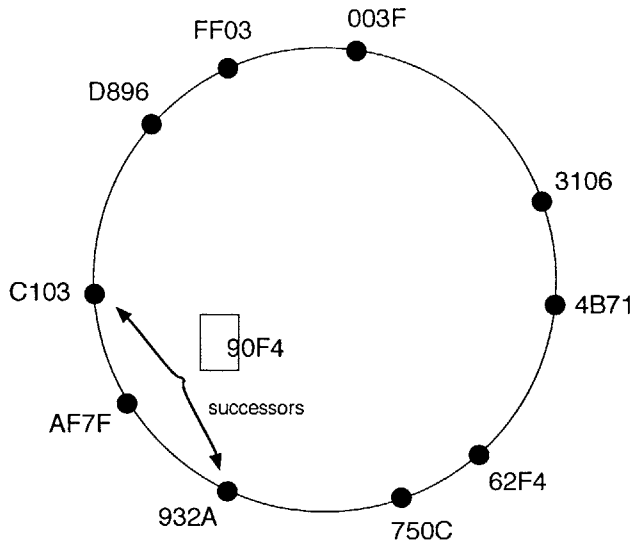

Figure 5-1: An example of an Aqua system with 10 hosts.

Figure 5-1 shows the circular nature of the Aqua ID space. There are 10 hosts in this Aqua system, each with an unique host ID. Block `90F4`'s $R = 3$ successors are `932A`, `AF7F`, and `C103`. The host and block IDs have abbreviated down to 16 bits.

5.3.3 Location Table

Knowing all the members of a DHT *a priori* simplifies locating the host that should store a block. For example, to find the IP address of a host that stores block with GID k, a host could look in its host list and find the host that is k's immediate successor.

In practice, not all of the hosts on the host list may be available at any given time. Each Aqua host keeps a *location table* that contains only hosts known to be alive. Initially, each host's location table contains only the host itself. The host periodically sends a *ping* message to each of the other hosts on the host list to determine if the host is reachable. The periodic probing as a message complexity of N^2, where N is the total number of hosts in the system. Additionally, whenever a host tries to contact another host to insert or retrieve data, the first host becomes aware of the second host's availability. When a host recovers from a failure and re-starts, the Aqua server on the host contacts all the hosts from the host list to notify them of the recovery.

Aqua delays its response to failed hosts to prevent excessive data movement in case of transient

failures [5]. If a host is suspected to be down, its entry is kept in the location table for T seconds (e.g. 30 minutes).

5.3.4 Semantics

Aqua stores/retrieves content-hash and public-key blocks in slightly different ways. When a host receives a `put(k,v)` for an immutable block, it looks in its location table and stores the block at the R successors of k, starting with the immediate successor. It reports success to the user after one store succeeds. On a `get(k)`, the host searches its location table and contacts some successors of k. If it cannot find the block among these hosts, it may contact additional hosts.

Aqua uses a simple quorum consensus protocol to store and retrieve mutable blocks. When a host receives a `put(k,v)` of a public-key block, it stores the block at the R successors (from location table) of k, starting with the immediate successor. Pastwatch reports success to the user after $1 + R/2$ stores succeed. On `get(k)`, a host retrieves $1 + R/2$ copies of the block among the R successors and returns the most recent block (i.e. by comparing the version number in the block) to the user. This protocol provides write-to-read consistency when the network does not partition and when only a few hosts fail.

In the presence of network partitions and many host failures, Aqua may not provide write-to-read consistency. The lack of strong consistency is not catastrophic. If Pastwatch fetches a stale copy of another user's mutable log-head, the worst thing that could happen is that the advisory locking algorithm no longer serializes users' commit operations. Unless concurrent commit operations by different users occur often, these temporary inconsistencies are not likely to produce forks on the branch tree. If Aqua's window of inconsistency is small, a user's new versions eventually appear in every user's branch tree.

Server Selection

With replication factor R, an Aqua server has multiple candidate hosts to fetch a block from. The technique of fetching data from the best candidate is typically called server selection. Aqua performs server section in the following fashion. Each Aqua host keeps exponentially weighted moving averages of the amount of time RPCs to each of the other hosts take. When the host receives a `get(k)` request for a content-hash block, it looks up in its location table to find the R successors

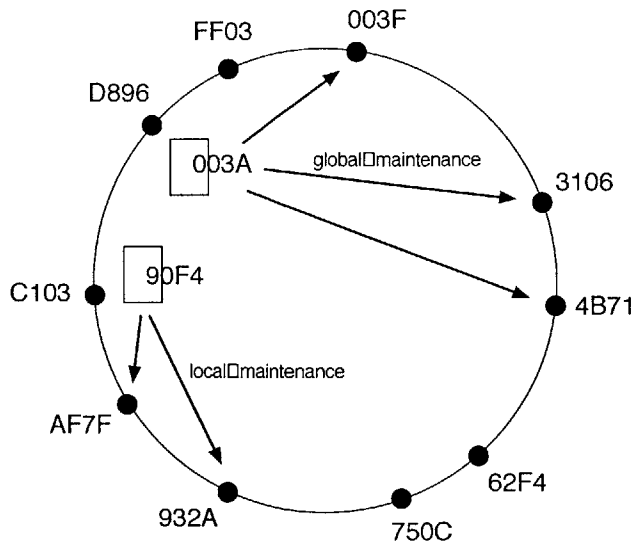

Figure 5-2: Aqua uses two replica maintenance protocols. Global maintenance moves blocks to the R successors of the block's GID. Local maintenance makes sure that there are R copies of each block on the R successors of the block's GID. In this example, $R = 3$.

of k, then picks J successors with the lowest RPC latencies to fetch the block from. When it obtains the first copy, it replies to the user. Fetching multiple copies of the block yields smaller latency when some successors have just failed but have not yet been evicted from the location table; contacting a failed host usually results in expensive timeouts.

5.3.5 Replica Maintenance

Aqua uses two replica management protocols to maintain R copies of each block on the block's R successors. They are variants of similar protocols from [9]. First, each host runs a *global maintenance protocol* that periodically moves blocks the host should not be storing to one of the block's successors, as shown in Figure 5-3. For example, in Figure 5-2 ($R = 3$), host `D896` should not be storing block `003A`. `global_maintenance` moves the block to the 3 successors of `003A`.

Second, each host runs a *local maintenance protocol* that periodically checks all the blocks it should be storing to make sure that R copies of each block exists, preferably on the R successors of the block. Figure 5-4 describes this protocol. It has three parts.

- First, `local_maintenance` determines the set of hosts that it could be sharing storage

```
1  global_maintenance(aquaId myID, int R) {
2    // myID is the local host's host ID, R is replication factor
3    aquaId a = myID;
4    while (1) {
5      // get Id of the local block that follows a
6      aquaId n = localblocks.nextId(a);
7      aquaId succs[] = loctable.lookup(n,R);
8      if (myID in succs) a = myID;  // should store the block
9      else {
10       // get all blocks between [n,succs[0]]
11       aquaBlock x[] = localblocks.getrange_inc(n,succs[0]);
12       bool moved = false;
13       for each s in succs {
14         bool ok = true;
15         for each aquaBlock b in x {
16           if ((s.get(b.id) == nil or stale) and  // s does not have latest b
17                !s.put(b.id, b))  // store b on s, fail if s is down
18             ok = false;
19         }
20         if (ok) moved = true;
21       }
22       if (moved) localblocks.delrange_inc(n,succs[0]);
23       a = succs[0];
24     }
25     sleep (GlobalMaintTimer);  // wait some time before running again
26   }
27 }
```

Figure 5-3: An Aqua host uses this protocol to move blocks to their appropriate replicas.

burdens with (line 5).

- Then, for each of these hosts, $preds[i+1]$, `local_maintenance` computes the set of replicas for a block (from the local host's stored blocks) that $preds[i+1]$ is an immediate successor of (line 8). Ideally, when all the hosts are available, this is just the R successors of that block. In case of transient failures – when a host fails but recovers within T seconds, the location table considers that host as a successor. Otherwise, if a would-be successor had failed and has not recovered within T seconds, `loctable.lookup` will return a new successor – this is a newly recruited replica.

- Finally, `local_maintenance` copies the appropriate set of blocks (line 10) to each of the replicas (lines 9-17).

For example, in Figure 5-2 ($R = 3$), when host `C103` runs `local_maintenance`, it makes sure that there are 3 copies of the block `90F4` on `90F4`'s 3 successors.

```
1  local_maintenance (aquaId myID, int R) {
2    // myID is the local host's host ID, R is replication factor
3    while (1) {
4      // the following returns R hosts that are predecessors of myID
5      aquaID preds[] = loctable.lookup_pred(myID,R);
6      preds.push_back (myID);
7      for (int i=0; i<R; i++) {
8        aquaID succs[] = loctable.lookup(preds[i+1],R);
9        // consider blocks whose GIDs are between (preds[i],preds[i+1]]
10       aquaBlock x[] = localblocks.getrange_rinc(preds[i],preds[i+1]);
11       for each s in succs {
12         for each aquaBlock b in x {
13           if (s.get(b.id) == nil or stale)
14             // s does not have latest b, store b on s, may fail if s is down
15             s.put(b.id, b);
16         }
17       }
18     }
19     sleep (LocalMaintTimer);  // wait some time before running again
20   }
21 }
```

Figure 5-4: An Aqua host uses this protocol to make sure there are R copies of each block on R successors of the block.

These replica maintenance strategies ensure that a block is available with high probability, in the absence of malicious hosts and when hosts fail independently. If Pastwatch inserts a block k into Aqua, and Aqua replicates the block on R different hosts, then the probability that the block is unavailable is $(1-\alpha)^R$, where α is the fraction of the time a single replica for the block is available. For example, for $R = 6$, to achieve 4 9's of availability, α is 78.5%.

Furthermore, because after a host becomes unreachable for more than T seconds Aqua recruits a new replica for the block, the amount of time a host is down and have a negative impact on availability (i.e. Aqua does not recruit a new replica to replace it) is bounded by T. Consider this to be MTTR. If T is 3600 seconds, for example, then for $\alpha = 0.785$, MTTF is 13411 seconds. This means as long as a host, and its network connection, can stay up for 13411 seconds at a time, and there are enough other hosts in the system that can replace the host as a replica, then Aqua offers 4 9's of availability.

[9] describes optimizations that make copying blocks between hosts efficient. Aqua implements these optimizations.

Command	Description
`put(id,block)`	Stores *block* under *id*
`get(id)`	Returns data stored under *id*
`put(id,xid,block)`	Stores *block* under *id*, on the host responsible for storing *xid*
`get(id,xid)`	Returns data whose GID is *id*, stored under *xid*
`checkpks(xid,K,V)`	Returns public-key blocks stored under *xid* with newer version numbers

Table 5.1: Each `aqua` program offers the following DHT interface. `past` communicates with an `aqua` program through RPC.

5.3.6 Fetching Many Log-head Blocks

A design goal of Pastwatch is to support projects with many members. This is challenging because each project member has a log, and Pastwatch needs to fetch all the log's log-head block on every user operation. Even if it fetches them in parallel, as the number of users increases, packet processing and bandwidth cost begin to dominate the cost of each user operation.

Pastwatch reduces the cost of fetching all the users' log-head blocks using the following protocol. First, Aqua offers a modified `put(id,xid,data)` interface that stores *data*, whose GID is *id*, on a host that is responsible for storing the GID *xid*. Pastwatch uses this interface to store the log-head blocks of all of a project's members on the same Aqua host, using the project's name (i.e. the GID of the block that stores the project's membership list) as the *xid* value. Second, Aqua offers a `checkpks(xid,K,V)` interface, where K is a list of public-key GIDs and V are their corresponding version numbers. When the client half of an `aqua` program receives a `checkpks` request, it forwards the request to the server storing *xid*. The server, then, returns public-key blocks from K whose corresponding version numbers in the server's database are higher than those in V. `past` uses this interface to fetch only the log-head blocks that have changed since the last time it contacted Aqua.

Figure 5.1 summarizes the DHT interface Aqua offers to applications.

5.4 Summary and Future Work

Aqua is on-going work. Current efforts focus on making Aqua more practical for deployment. For example, in practice, each host may contribute different amount of network and storage resources. This means load balancing is needed in order to effectively use all the contributed resources. Load

balancing is challenging. First, a host is responsible for the address segment between the host's ID and the ID of the host's Rth predecessor. This means hosts that have less resources to contribute should not have IDs that are next to hosts with more resources. Another load balancing goal is to gracefully and effectively add new resources to the system. When a new host joins the system, re-assigning identifiers of all the hosts is not practical. Simply assigning the new host a unique identifier, hence distributing an existing host's responsibility to two hosts, may not put the new resources to effective use.

Another interesting problem is that a public Aqua system may contain many hosts, and different hosts may want to support a different set of projects. Users of a Pastwatch project should be able to find and use the set of hosts that are willing to store data for the project.

Notes

[4] www.sleepycat.com

Chapter 6

Performance

This chapter describes Pastwatch's performance. The main questions it answers are 1) is Pastwatch usable? 2) can it support a large number of users despite the use of per-user logging? 3) how long does it take to check out a project's repository after many users have committed changes into the repository?

This chapter shows that Pastwatch performs well over the wide-area network; its performance is comparable to using CVS over the Internet. This means Pastwatch is able to provide availability and avoid single points of failures and trusts without compromising its performance.

6.1 Experiment Setup

This chapter evaluates Pastwatch's performance over the Internet, and compares its performance with that of CVS [4]. Each Pastwatch experiment uses an 8-host Aqua system that consists of mostly RON [2] hosts. Each host is responsible for exactly 12.5% of the total address space. Aqua replicates each block 6 times. For content-hash blocks, an Aqua host tries to fetch 3 copies of the block and returns when a single copy is received. Each host retrieves blocks from hosts with the lowest measured RPC latencies. Table 6.1 shows round-trip latencies between these hosts.

There are two client hosts, not part of the Aqua system. Each Pastwatch or CVS experiment issues repository operations from these two hosts. `bos` is a 1.6 GHz AMD computer with 1 GB RAM and a 160 MB/s SCSI disk. `ana` is a 1.7 GHz Intel Pentium 4 computer with 512 MB RAM and also a 160 MB/s SCSI disk. Most of the experiments require users to contact `nyu` and `dwest`.

Hosts	mit	nortel	ucsd	nyu	cmu	utah	dwest	chic
mit	-	19 ms	85 ms	8 ms	20 ms	60 ms	82 ms	29 ms
nortel		-	88 ms	37 ms	47 ms	75 ms	75 ms	20 ms
ucsd			-	81 ms	80 ms	45 ms	8 ms	63 ms
nyu				-	15 ms	57 ms	71 ms	27 ms
cmu					-	68 ms	81 ms	39 ms
utah						-	85 ms	34 ms
dwest							-	61 ms
chic								-

Table 6.1: Best case round-trip latency between hosts in the DHT.

The best case round-trip latency between bos and nyu is 8 ms, between ana and nyu is 66 ms, and between ana and dwest is 6 ms. Clients and Aqua servers communicate using TCP. The best measured (using ttcp) bandwidth for one TCP flow from bos to nyu is 29 Mbps, from nyu to bos is 1.2 Mbps, from ana to nyu is 0.2 Mbps, from nyu to ana is 0.4 Mbps, from ana to dwest is 0.3 Mbps, and from dwest to ana is 13 Mbps.

Unless otherwise stated, each experiment uses a trace of 40 commit operations taken from the CVS log of an open-source project. At the start of the experiment, a user imports 681 source-code files and directories into the repository. Subsequently, users check out the shared files into their own working directories, replay the commit operations, and update their working directories to merge in new changes. The average number of files each commit operation changes is 4.8. The standard deviation is 6.2, the median is 3, and the highest is 51. In total, the 40 commit operations modify roughly 4330 lines in the source-code and add 6 new files.

6.2 Basic Performance

This section compares the costs of basic Pastwatch repository operations with similar CVS operations. It evaluates four operations: importing a set of files, checking out a set of files, committing changes, and updating a working directory.

Each experiment involves two users. For simplicity, this chapter refers to each user by the name of the host the user runs repository operations on. In each experiment, bos imports a set of files into an empty repository. Both users (bos and ana) then check out a copy of the files onto their own computers. Afterward, during each iteration of the experiment, one user commits some changes to

Trials	bos				ana		
	im	**co**	**20 ci**	**20 up**	**co**	**20 ci**	**20 up**
CVS	15.9	6.0	33.5	33.2	121.2	81.4	94.3
Pastwatch/1srv	75.8	2.2	39.6	29.1	121.0	45.5	41.9
Pastwatch/Aqua	114.3	2.2	49.4	34.1	23.4	47.9	36.2

Table 6.2: Runtime, in seconds, of Pastwatch and CVS import (im), check out (co), commit (ci), and update (up) commands. Users on bos and anarun these commands. In the CVS experiment, users contact the CVS server running on nyu. In the Pastwatch/1srv experiment, users contact a single Aqua server running on nyu. In the Pastwatch/Aqua experiment, each user contacts a host from an 8-host DHT: bos contacts nyu, while ana contacts dwest. The costs of update and commit are sums over 20 update and commit operations by each user. Each value is the median of 20 runs.

the repository and the other user updates that user's working directory to retrieve the changes. In the subsequent iteration, the second user commits changes and the first user retrieves those changes. In total, each user commits changes 20 times and updates a working directory 20 times, although the two users perform different commit and update operations that affect different number of files. Both Pastwatch and CVS experiments use this workload. For Pastwatch, the number of users in the membership list is two. In all experiments, the file system that the experiment uses on each machine has soft-update [15] turned on.

Table 6.2 reports the costs, in seconds, of an import operation (im) and a checkout operation (co), and the total costs of 20 commit (ci) and update (up) operations on each host. Each number is median of 20 runs. In the CVS experiment, users contact a CVS server on nyu. The next experiment, Pastwatch/1srv, evaluates Pastwatch without using a DHT. Each user contacts a single Aqua server, running on nyu, instead of the DHT. Finally, in the Pastwatch/Aqua experiment, users contact the 8-host Aqua DHT described earlier.

To import new files into a repository, using Pastwatch takes much longer than CVS. Each Aqua server stores data in a db3 database that uses logging and transactions, rather than in a file system. This means Aqua not only stores more data than CVS, due to the overhead of using db3, but also spends more time writing data. Pastwatch/Aqua has an even higher cost because in this case, Aqua replicates each block 6 times. These reasons also explain the differences in commit costs between the three experiments on bos.

Checking out the repository's files on bos is very fast, as shown by the second column in Table 6.2, because the Pastwatch software only fetches data from a snapshot in the local branch tree

– the previous import operation had created a snapshot in the local tree that contains all the new files. On the other hand, CVS on `bos` must contact the server at `nyu` to download files.

Because `ana` is using the repository for the first time, the checkout operations in both Pastwatch experiments must fetch data over the network to create local branch trees. The available bandwidth between `ana` and `nyu`is low. The costs of CVS and Pastwatch/1srv checkout operations are bandwidth limited. Each experiment downloads about 5 megabytes of data. In the Pastwatch/1srv case, Pastwatch spends around 110 seconds fetching commit records and delta blocks from the Aqua server on `nyu`. This yields 0.36 Mbps. The measured best case available bandwidth from `nyu` to `ana`is 0.4 Mbps. On the other hand, in the Pastwatch/Aqua experiment, `ana` may contact any of the 8 Aqua hosts. It chooses `dwest`, the closest one. Furthermore, `dwest` also contacts the closest replicas of each block when it receives a `get` request. The use of nearby servers explains why the Pastwatch/Aqua checkout operation on `ana` outperforms CVS and Pastwatch/1srv significantly.

The network latency between `ana` and `nyu` is high. CVS does not perform well in this case because each operation requires multiple network round-trips to compare the files on the client and on the server. On the other hand, each Pastwatch update operation only contacts the DHT once to fetch new changes, then completes the rest of the operation entirely using a local branch tree. Similarly, each Pastwatch commit operation contacts the DHT once to check if there are any new changes, then contact the DHT once more at the end of the operation to insert new commit record and delta blocks in parallel. Reducing the number of network round-trips, in this case, improves performance.

Again, `ana` must contact the distant `nyu` server in the Pastwatch/1srv experiment, hence the total cost of the 20 update operations on `ana`, in this case, is higher than in the Pastwatch/Aqua experiment. On the other hand, because in the Pastwatch/Aqua experiment `dwest` must replicate each block it receives from `ana` 6 times, the total cost of the 20 commit operations is higher than in the Pastwatch/1srv experiment, despite server locality.

Table 6.3 breaks down the cost of each commit and update operation on `ana`, in the Pastwatch/Aqua experiment. Each number in the table is a median value over 20 commit or update operations, then over 20 runs of each experiment. The cost of inserting data into the DHT clearly dominates the cost of a commit operation. On the other hand, retrieving changes from the DHT and building snapshots that capture those changes account for 31% of the total cost of an update

Tasks	Pastwatch/Aqua	
	commit	update
Fetch view block	234 ms	231 ms
Acquire lock/build snapshots	517 ms	492 ms
Insert changes or update working dir	1293 ms	681 ms
Update working dir's metadata	190 ms	241 ms
Total	2.2 s	1.6 s

Table 6.3: Time spent (median) in each step of a commit/update operation on `ana`.

operation. Comparing files in the working directory against those in a snapshot accounts for 43%.

Storage Cost

At the end of each run of a Pastwatch/1srv experiment, the Aqua server's database is 14 megabytes, storing 4.7 megabytes of application data in 3,192 blocks. Database storage overhead accounts for the remaining 9.3 megabytes. On each client's machine, the size of the Pastwatch's database is 31 megabytes, with 7.7 megabytes of application data in 4,534 blocks. These blocks include both blocks used for snapshots and the 3,192 blocks from the Aqua server. Pastwatch built 42 snapshots for each user. The fact that all the snapshots only use 1,342 blocks shows that snapshots share many of their blocks. Also, recall that some snapshots may reuse data blocks that `Write` deltas point to.

Because each Aqua host is responsible for exactly 12.5% of the Aqua's address space, they share the same storage burden. At the end of a run of the Pastwatch/Aqua experiment, the total size of all the databases on all the Aqua hosts is 122 megabytes, or 15 megabytes per host. Each host's database contains, on average, 3.6 megabytes of application data, in 2,391 blocks. The average size of each block is 1,477 bytes, with a median of only 174 bytes. For the same workload, the Pastwatch/1srv experiment inserts 4.7 megabytes of application data. With replication, 6 times of 4.7 is close to the 28.8 megabytes of total application data on all the Aqua hosts. In comparison, the CVS repository on `nyu` is 5.2 megabytes after each run of the CVS experiment.

Summary

This section demonstrates that Pastwatch's performance is comparable to that of CVS over the wide-area network. More specifically, the round-trip latency between a client and a server that the client

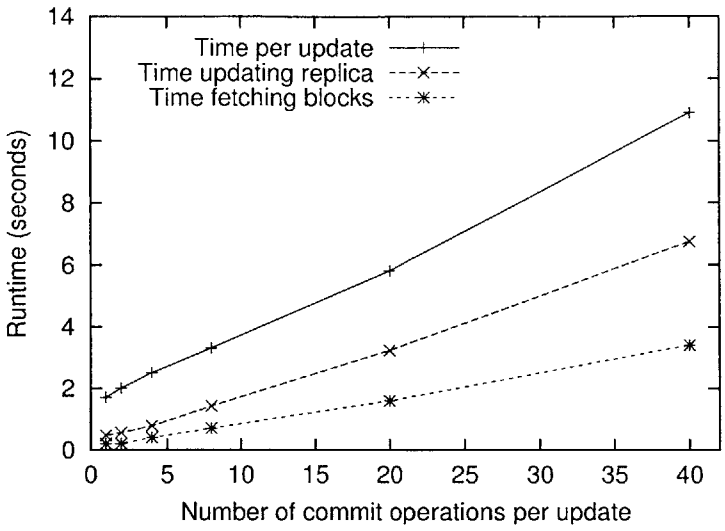

Figure 6-1: Average runtime of an update operation as the number of commit operations per update operation increases. Only one user commits changes into the repository.

talk to is low, Pastwatch and CVS have very similar performance. A CVS commit operation is slightly faster because CVS does not replicate data. When the round-trip latency between a client and a server that the client talk to is high, Pastwatch performs better than CVS because each of its repository operations requires at most a few network round-trips, completing most tasks using the local branch tree. Furthermore, an Aqua system has multiple hosts, so Pastwatch can pick a closer server to talk to.

6.3 Retrieving Many Changes

This section examines the cost of updating a working directory to retrieve multiple changes from another user. Each update operation needs to fetch multiple commit records, build multiple snapshots, and merge multiple sets of changes into the user's working directory.

Each experiment uses the 8-host Aqua system from Section 6.2. The workload is slightly different: the user on `bos` commits all 40 sets of changes into the repository; it commits several changes before the user on `ana`performs an update operation. The number of commits per update operation varies for each experiment. Again, `bos` contacts the Aqua server on `nyu`, and `ana` contacts the Aqua server on `dwest`.

Figure 6-1 reports the average runtime of update operations, as the number of commits per update increases. Each point on the graph is an average over 10 runs of an experiment. For example, with 8 commit operations per update operation, the user on `ana` updates the local working directory 5 times per experiment. Over the course of 10 runs of the experiment, the average runtime over 50 update operations is shown on the graph.

The top curve in Figure 6-1 shows that the cost of an update operation increases linearly with the number of commit record it needs to fetch. The increase in runtime is mainly due to that Pastwatch needs to spend more time fetching commit records and delta blocks and building new snapshots. For example, the middle curve in the figure shows the time spent fetching these blocks and building snapshots. The bottom curve in the figure shows only the time spent fetching blocks.

6.4 Many Users

A concern with using per-user logging is that as the number of project members increases, the cost of scanning all their logs also increases. This section shows that while this is true, Pastwatch's implementation keeps the extra cost per user low enough to support a large number of users.

The first set of experiments reports Pastwatch's performance as the number of members of a project increases, but limiting the number of members who commit changes to two. In practice, a project may have many members, but often only a few of them actively make changes to the project's files. These experiments reflect this scenario.

Each experiment uses the 8-host DHT setup and workload from Section 6.2. Two users alternately commit changes and retrieve each other's changes. During each run of an experiment, each user runs 20 commit and 20 update operations. On each operation, Pastwatch, on behalf of a user, uses the `checkpks` RPC to determine which log-heads have changed since the last time it updated the user's local tree, and only fetches those log-heads and their attached commit records. Each run of an experiment starts with a new repository.

Figure 6-2 shows that the total costs of commit and update operations on `bos` and `ana` increase slightly as the number of project members increases. Each value is the median of 20 runs of each experiment. Each `checkpks` RPC message uses 24 bytes per project member, so as the number of members increases, so does the size of each `checkpks` RPC message. Also, for each commit

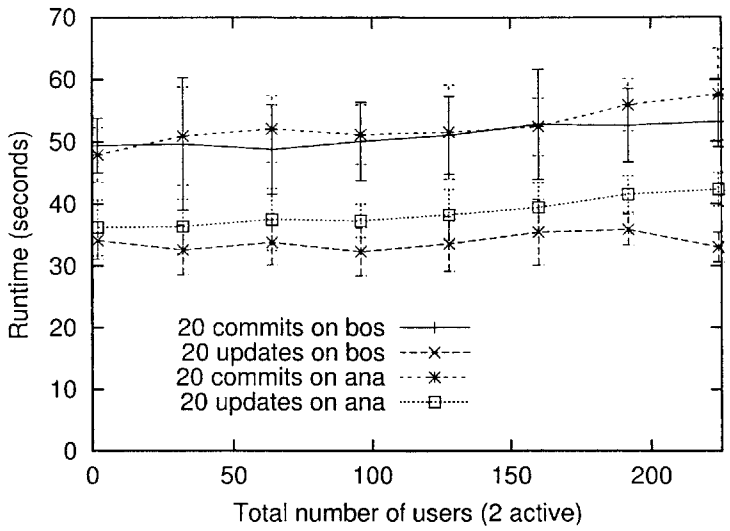

Figure 6-2: Runtime of Pastwatch commit and update operations as the number of total project members increases. Only two members commit changes into the repository. Each value is the median of 20 runs. The error bars are standard deviations.

operation, Pastwatch iterates over the log-heads of all the members to check for lock contention (see Section 3.6). These factors increase the cost of each repository operation.

The standard deviation for each value in Figure 6-2 is between 3 and 10 seconds. Varying network conditions and the fact that different hosts receive and process the `checkpks` RPCs cause each run of the same experiment (i.e. with the same number of project members) to complete in different amounts of time. Recall that Aqua forwards each `checkpks` RPC to the host that stores the project's membership list. For each run of an experiment, this is a different host, with different round-trip latencies and available bandwidth to the other hosts. Each update operation in one run of an experiment sends one `checkpks` RPC to this host. Each commit operation in one run of an experiment sends two `checkpks` RPCs to this host.

The next set of experiments examines the effects of multiple users committing changes into the repository. These experiments use the setup and workload from Section 6.3, except instead of one user committing all the changes on one machine, multiple users do. In each experiment, x number of project members commit changes on `bos`. After they each commit a set of changes, a user on `ana` updates the local working directory to retrieve these x changes, from x different logs. Each project has $x + 1$ project members.

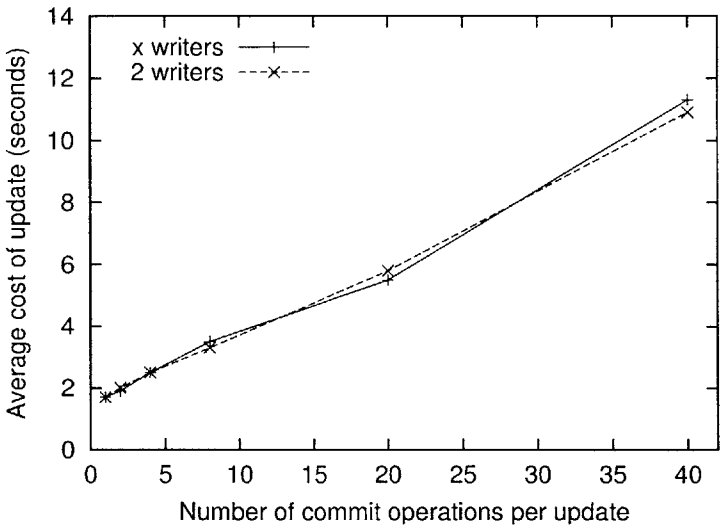

Figure 6-3: Average runtime of an update operation as the number of commit operations per update operation increases. Each commit operation is performed by a different user.

Figure 6-3 compares the cost of update operations when one user commits changes to when multiple users commit changes. They are similar, suggesting that scanning multiple logs does not increase the cost of each operation.

Chapter 7

Related Work

Pastwatch is motivated by the popularity of SourceForge.net[5], a CVS-repository hosting service for open-source projects. A similar service is Savannah[6]. Both services remove users' burden of maintaining dedicated servers. Unlike Pastwatch, they are centralized systems.

Pastwatch's cooperative approach is motivated by recent work on peer-to-peer storage, particularly FreeNet [10], Past [37], and CFS [11, 12]. The data authentication mechanisms in these systems limit them to read-only data; once created by the original publisher, the data cannot be changed. CFS builds a read-only file system on top of peer-to-peer storage, using ideas from SFSRO [14]. Ivy [29] differs from these systems in that it uses per-user logs to implement a multi-user read/write file system on top of peer-to-peer storage.

Pastwatch's main contribution relative to these systems is that it uses the peer-to-peer storage to maintain soft-state. Pastwatch is therefore more robust with respect to the lack of strong consistency and the untrusted nature of these peer-to-peer storage systems.

Pastwatch's user interface is influenced by CVS [4]. Unlike CVS, Pastwatch allows users to commit changes while disconnected. Several other version control systems, such as BitKeeper[7], Arch[8], Monotone[9], and Subversion[10], also support disconnected operations. These systems, however, require a central server that stores users' changes. Unlike these systems, the main focus of Pastwatch's design is not on providing rich version control features, but rather on availability and avoiding dedicated servers.

At the expense of consistency, Monotone allows users to exchange their changes over e-mail or NNTP news service. This approach means users do not have to maintain dedicated servers.

Pastwatch provides this property without sacrificing consistency in the common case that the DHT provides write-to-read consistency.

Like Pastwatch, many version control systems use branch trees to capture multiple lines of development and, in the case of disconnected operations, implicit branching. Each Pastwatch user's branch tree additionally reveals when the DHT has behaved in an inconsistent manner (e.g. hiding new versions from users).

An alternative to running CVS on a dedicated server is to use it on a serverless file system, such as Ivy [29]. Ivy presents a single file system image that appears much like an NFS file system. In contrast to NFS, Ivy does not require a dedicated server; instead, it stores all data and meta-data in a peer-to-peer block storage system. The storage system replicates blocks, giving Ivy the potential to be highly available. The drawback of using Ivy is that the file system interface does not support atomic multi-step operations; if a user crashes or disconnects while modifying a repository's files, the repository could be left in an inconsistent state. Ivy is also slow; every file system operation must fetch data from potentially distant servers. Furthermore, if two disconnected users modify the same files or directories in a repository, after they reconnect, the affected file or directory may be corrupted. From a version control system's point of view, however, these changes should just result in two different versions of the shared files. A user can then examine and merge the two versions at any time.

7.1 Logging

Sprite LFS [36] represents a file system as a log of operations, along with a snapshot of i-number to i-node location mappings. LFS uses a single log managed by a single server in order to to speed up small write performance. Pastwatch uses multiple logs to let multiple users commit changes without a central lock server; it does not gain any performance by use of logs.

Per-user logging was developed for the Ivy [29] peer-to-peer read/write file system. However, whereas Ivy represents a file system as a set of logs and scans every log to answer each file system request, Pastwatch uses logs to communicate changes that produce new, immutable versions of the shared files. A user checks out shared files from a local branch tree, instead of directly from the logs. Ivy also stores a version vector in each log record and uses the version vector to merge all

the log records into one single virtual log. In contrast, Pastwatch stores a single reference in each log record and uses the reference to build a tree. Whereas a version vector's size, and hence the size of each Ivy log record, grows linearly with the number of users, each Pastwatch log record's size remains constant as the number of users in a project grows. Consequently, Pastwatch is able to support a large number of users per project.

7.2 Disconnected Semantics and Conflict Resolution

Coda also uses logs to capture partitioned operations [39] and disconnected operations [21]. For example, a Coda client keeps a replay log that records modifications to the client's local copies while the client is in disconnected mode. When the client reconnects with the server, Coda propagates client's changes to the server by replaying the log on the server. Coda detects changes that conflict with changes made by other users, and presents the details of the changes to application-specific conflict resolvers. Pastwatch similarly uses logs to represent partitioned and disconnected changes. Unlike Coda, Pastwatch only automatically merge non-conflicting changes from different partitions upon user request, to avoid leaving the repository in a state where only a subset of a coherent set of changes is visible.

Bayou [44, 31] uses logs to communicate different users' changes to a shared database. Each host maintains a log of all the changes it knows about, including its own and those by other hosts. Hosts merge these logs pairwise when they talk to each other. Pastwatch borrows the idea of using logs to propagate changes to a shared data structure, but differs from Bayou in several important ways. First, Pastwatch users do not have to depend on a primary host to decide on the final ordering of all the changes; the fact that versions are immutable and each depends on a parent lets users independently arrive at the same branch trees given the same set of versions. Second, Pastwatch stores log records in a DHT, rather than exchanging log records pairwise among different users. This means users could find each other's changes even if they come on-line at different times. In the common case that the DHT behaves in an atomic manner, Pastwatch also provides strong consistency for connected users: the effects of a commit operation becomes immediately visible to the other users. Finally, whereas Bayou requires a trusted entity to detect users hiding changes from each other [40], Pastwatch users could detect these attacks more easily.

Ficus [30] is a distributed file system in which any replica can be updated. Ficus automatically merges non-conflicting updates from different replicas, and uses version vectors to detect conflicting updates and to signal them to the user. Pastwatch also faces the problem of conflicting updates performed in different network partitions, but unlike Bayou, Coda, and Ficus, it does not automatically merge them.

Users of a Groove [32] system share read/write data by storing a local copy and exchanging update messages. Pastwatch users communicate indirectly via DHash storage, instead of directly among each other. This enables Pastwatch to scale better with an increasing number of read-only participants. Groove also stores the updates in a log file at a centralized server, where offline users can fetch when they come back online. Pastwatch does not depend on a central, dedicated server.

7.3 Storing Data on Untrusted Servers

BFS [8], OceanStore [22], and Farsite [1] all store data on untrusted servers using Castro and Liskov's practical Byzantine agreement algorithm [8]. Multiple clients are allowed to modify a given data item; they do this by sending update operations to a small group of servers holding replicas of the data. These servers agree on which operations to apply, and in what order, using Byzantine agreement. The reason Byzantine agreement is needed is that clients cannot directly validate the data they fetch from the servers, since the data may be the result of incremental operations that no one client is aware of.

SUNDR [28] uses fork-consistency to detect stale-data attacks. If a server hides changes from a user, then the user can no longer see any changes by other users ever again. In this case, off-of-band communication between the users reveals the attack. Using a branch tree, Pastwatch provides similar properties with respect to an untrusted DHT, although it is sometimes difficult to distinguish between a network partition and a stale-data attack.

TDB [25], S4 [43], and PFS [41] use logging and (for TDB and PFS) collision-resistant hashes to allow modifications by malicious users or corrupted storage devices to be detected and (with S4) undone; Pastwatch uses similar techniques.

Spreitzer et al. [40] suggest ways to use cryptographically signed log entries to prevent servers from tampering with client updates or producing inconsistent log orderings; this is in the context

of Bayou-like systems. Pastwatch's logs are simpler than Bayou's, since only one client writes any given log. This allows Pastwatch to protect log integrity, despite untrusted DHash servers, by relatively simple per-client use of cryptographic hashes and public key signatures. Pastwatch additionally uses a branch tree to expose stale-data attacks.

7.4 Distributed Storage

Pastwatch is layered on top of a distributed hash table (DHT) [11, 12, 16, 26, 33, 34, 38, 48]. Using a DHT avoids single points of failures and provides availability. Although some DHTs use quorum and BFT techniques to provide strong semantics [24, 35], Pastwatch does not rely on the DHT to provide strong consistency and be trustworthy.

Zebra [17] maintains a per-client log of file contents, striped across multiple network nodes. Zebra serializes meta-data operations through a single meta-data server. Pastwatch and Ivy borrow the idea of per-client logs, but extends them to meta-data as well as file contents.

xFS [3], the Serverless Network File System, distributes both data and meta-data across participating hosts. For every piece of meta-data (e.g. an i-node) there is a host that is responsible for serializing updates to that meta-data to maintain consistency. Pastwatch avoids using single dedicated servers, but does not provide strong consistency.

Frangipani [45] is a distributed file system with two layers: a distributed storage service that acts as a virtual disk and a set of symmetric file servers. Frangipani maintains fairly conventional on-disk file system structures, with small, per-server meta-data logs to improve performance and recoverability. Frangipani servers use locks to serialize updates to meta-data. This approach requires reliable and trustworthy servers.

Harp [23] uses a primary copy scheme to maintain identical replicas of the entire file system. Clients send all NFS requests to the current primary server, which serializes them. A Harp system consists of a small cluster of well managed servers, probably physically co-located. Pastwatch works without any central cluster of dedicated servers – at the expense of strong consistency.

Notes

[5] www.sourceforge.net

[6] savannah.gnu.org,savannah.nongnu.org

[7] www.bitkeeper.com
[8] wiki.gnuarch.org
[9] www.venge.net/monotone
[10] subversion.tigris.org

Chapter 8

Conclusion

This thesis describes Pastwatch, a distributed version control system. Pastwatch maintains versions of users' shared files. Each version is immutable: to make changes, a user checks out a version onto the user's computer, edits the files locally, then commits the changes to create a new version. The motivation behind Pastwatch is to support wide-area read/write file sharing. An example of this type of sharing is when loosely affiliated programmers from different parts of the world collaborate to work on open-source software projects.

Pastwatch offers the following useful properties: it supports disconnected operation, it makes shared files highly available, and it does not require dedicated servers.

Pastwatch provides these properties using two interacting approaches. First, it maintains a local branch tree of versions on each user's computer. A user can check out versions from the local tree and commit changes into the local tree. Second, Pastwatch uses a shared branch tree in a DHT to publish users' new versions. It contacts the tree to keep a user's local branch tree up-to-date.

This thesis has several contributions.

- It describes how to provide eventual consistency for an optimistically replicated object (i.e. a project's shared files) using branch trees. Each user organizes every users' changes to the object into a local branch tree. Each node on the tree is an immutable version of the object, refers to its parent, and contains exactly the result of applying a set of changes to the parent's contents. Given the same set of user changes, every user independently constructs the same branch tree.

- It uses forks on a branch tree to reveal when a user made changes to a stale local replica and created a new version. This could occur when some users are disconnected or when the propagation mechanism does not make a user's new version visible to other users immediately. Pastwatch also helps users detect and recover from forking.

- It describes how to build a single-writer, multi-reader data structure (i.e. the branch tree) in the DHT using per-user logging. Each user appends changes to the data structure (i.e. new versions) to the user's own log and scans all the logs to reconstruct the data structure. This arrangement is attractive because most DHTs require mutable blocks to be cryptographically signed and users may not want to share private-keys.

- It uses forks on a branch tree to reveal when a DHT behaves in an inconsistent manner, such as returning stale data or no data to users. These inconsistencies could be caused by long network delays, host failures, networking partitioning, or malicious DHT hosts.

Bibliography

[1] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[2] D. Andersen, H. Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of the ACM Symposium on Operating System Principles*, October 2001.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the ACM Symposium on Operating System Principles*, December 1995.

[4] B. Berliner. CVS II: Parallelizing software development. In *Proc. Winter 1990 USENIX Technical Conference*, 1990.

[5] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2003.

[6] J. Burns and N. Lynch. Mutual exclusion using indivisible reads and writes. In *In Proceedings of 18th Annual Allerton Conference on Communications, Control, and Computing*, 1980.

[7] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[8] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

[9] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.

[10] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[11] F. Dabek, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating System Principles*, October 2001.

[12] F. Dabek, J. Li, E. Sit, J. Robertson, M. Frans Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, March 2004.

[13] FIPS 180-1. *Secure Hash Standard.* U.S. Department of Commerce/N.I.S.T., National Technical Information Service, April 1995.

[14] K. Fu, M. Frans Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, October 2000.

[15] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2), 2000.

[16] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[17] J. Hartman and J. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3), 1995.

[18] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 13(1), January 1991.

[19] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed cachine protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1997.

[20] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Adoption of dhts with openhash, a public dht service. In *In Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, February 2004.

[21] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the ACM Symposium on Operating System Principles*, 1991.

[22] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. of ACM ASPLOS*, November 2000.

[23] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of the ACM Symposium on Operating System Principles*, 1991.

[24] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, Toulouse, France, October 2002.

[25] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, October 2000.

[26] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st IPTPS*, March 2002.

[27] D. Mazières. A toolkit for user-level file systems. In *Proc. of the Usenix Technical Conference*, June 2001.

[28] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, July 2002.

[29] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[30] T. Page, R. Guy, G. Popek, and J. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report UCLA-CSD 910005, 1991.

[31] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the ACM Symposium on Operating System Principles*, October 1997.

[32] Bill Pitzer. *Using Groove 2.0.* Que Publishing, 2002.

[33] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, August 2001.

[34] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of USENIX File and Storage Technologies*, March 2003.

[35] R. Rodrigues, B. Liskov, and L. Shrira. The design of a robust peer-to-peer system. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[36] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), 1992.

[37] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the ACM Symposium on Operating System Principles*, October 2001.

[38] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[39] M. Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems*, 20(2), 2002.

[40] M. Spreitzer, M. Theimer, K. Petersen, A. Demers, and D. Terry. Dealing with server corruption in weakly consistent, replicated data systems. In *Proc. of the ACM/IEEE MobiCom Conference*, September 1997.

[41] C. Stein, J. Howard, and M. Seltzer. Unifying file system protection. In *Proc. of the USENIX Technical Conference*, 2001.

[42] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, August 2001.

[43] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, and G. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, October 2000.

[44] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the ACM Symposium on Operating System Principles*, December 1995.

[45] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the ACM Symposium on Operating System Principles*, 1997.

[46] H. Williams. A modification of the RSA public-key encryption procedure. In *IEEE Transactions on Information Theory*, volume IT-26(6), November 1980.

[47] T. Ylönen. SSH - secure login connections over the internet. In *Proceedings of the 6th Usenix Security Symposium*, July 1996.

[48] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.